Contrôle de la propagation et de la recherche dans un solveur de contraintes

Charles Prud'Homme

l'université nantes angers le mans
PÔLE DE RECHERCHE ET D'ENSEIGNEMENT SUPÉRIEUR

MINES Nantes
Institut Mines-Télécom

Thèse de Doctorat

Charles PRUD'HOMME

Mémoire présenté en vue de l'obtention du
grade de Docteur de l'École nationale supérieure des mines de Nantes
sous le label de l'Université de Nantes Angers Le Mans

École doctorale : Sciences et technologies de l'information, et mathématiques

Discipline : Informatique et applications, section CNU 27
Unité de recherche : Laboratoire d'informatique de Nantes-Atlantique (LINA)

Soutenue le 28 février 2014
Thèse n° : 2014 EMNA0194

Contrôle de la Propagation et de la Recherche dans un Solveur de Contraintes

JURY

Président :	**M. Gérard VERFAILLIE**, Directeur de Recherche, ONERA
Rapporteurs :	**M. Jean-Charles RÉGIN**, Professeur, Université de Nice **M. Gérard VERFAILLIE**, Directeur de Recherche, ONERA
Examinateur :	**M. Benoît ROTTEMBOURG**, Directeur Associé, Eurodécision
Directeur de thèse :	**M. Narendra JUSSIEN**, Professeur, École des Mines de Nantes
Co-directeurs de thèse :	**M. Xavier LORCA**, Maitre-assistant, École des Mines de Nantes **M. Rémi DOUENCE**, Maitre-assistant, École des Mines de Nantes

«Ce que j'ai vu ce soir là justifie toutes les tortures.
Même si la pire des tortures fut de ne pas en voir plus...»
(OSS 117 : Le Caire nid d'espions)

Remerciements

« 夜になると、暗くなります。»
Yoru ni naru to, kuraku narimasu.
Quand la nuit tombe, il fait plus sombre.
(Proverbe japonais)

Les travaux qui présentés dans cette thèse ont été menés au sein de l'équipe TASC du département Informatique de l'École des Mines de Nantes. Je tiens à remercier mes encadrants avec qui les relations ne se sont pas limitées au cadre professionnel. Rémi Douence, avec lequel j'ai commencé par faire des court-métrages avant de me lancer dans ce "très long-métrage". Xavier Lorca qui a toujours su prendre le recul nécessaire pour me conseiller et avancer, souvent bien plus vite que moi. Et, Narendra Jussien, à qui je dois d'être l'ingénieur Choco et bien plus encore.

Il est clair que les membres du département Informatique et de l'équipe TASC (passés et présents), ont contribué à ce que cette thèse se déroule dans les meilleures conditions. Merci à vous pour votre humeur (bonne ou mauvaise), vos discussions (scientifique ou non) et les pauses café. Merci plus particulièrement à Jean-Guillaume Fages pour la bonne compagnie, la motivation et le code. L'article que nous avons co-rédigé sonnait comme une chanson d'Amy Winehouse. Je remercie aussi Laurent Perron pour les nombreuses discussions que nous avons pu avoir à propos de la conception de solveurs. Il a certainement plus participé à Choco qu'il ne le soupçonne.

Mon souhait de faire une thèse remonte à plusieurs années, et Christelle Guéret, Nubia Velasco, et avant elles, Eric Pinson et Laurent Péridy ont leur part de responsabilités.

Je tiens à remercier les membres du jury pour avoir bien voulu y participer et plus particulièrement Messieurs Jean-Charles Régin et Gérard Verfaillie pour avoir accepté de suivre ma thèse et d'en être les rapporteurs.

Tout cela ne serait rien sans ma famille et mes amis : merci à ma mère, mon père, Dorothée (tu es la suivante), Pierre, Alain, Marie-Hélène, Marion, Ronan, Baptiste, Delphine, GuiDavHelAd, Anne, Mickey, Nono, Yohann, ... Il faut aussi décrocher par moment.

Enfin, les plus importantes dans cette équation, mes drôles de dames : Emilie, Clémence et Amandine. J'ai commencé cette thèse en fêtant la naissance de Clémence et je la termine en fêtant celle d'Amandine. Mais c'est sans surprise ma khaleesi qui m'a le plus soutenu ces trois dernières années. Je te suis éternellement, voire un peu plus, reconnaissant d'avoir été à mes côtés.

Table des matières

Liste des tableaux

Table des figures

Liste des Algorithmes

1

Introduction

La modélisation de problème sous forme de problème de satisfaction de contraintes, ou CSP (pour "*Constraint Satisfaction Problem*" en anglais), et leur résolution, a vu naitre au début des années 80 une nouvelle discipline d'aide à la décision : la *Programmation Par Contraintes*, ou PPC. Ce paradigme de programmation emprunte ses caractéristiques à divers domaines. Sa déclarativité trouve son origine dans le Calcul Formel, elle se sert des algorithmes de la Recherche Opérationnelle et des Mathématiques Discrètes et s'inspire des méthodes d'exploration de l'Intelligence Artificielle. Très tôt, les industriels, tels que Ilog, Cosytec et aujourd'hui Google, ont montré un intérêt pour la programmation par contraintes qui fait désormais partie du portfolio de solutions globales de traitement de problèmes combinatoires réels. En effet, cette technique ne s'attache pas à traiter un type de problème en particulier, mais fournit une librairie d'instruments capables de traiter une large gamme de problèmes combinatoires. Ces outils ont été conçus pour permettre à des non-spécialistes d'aborder ces problèmes tant stratégiques qu'opérationnels, parmi lesquels on trouve des problèmes de planification, d'ordonnancement, de logistique, l'analyse financière ou encore la bio-informatique. La programmation par contraintes se distingue d'autres méthodes de Recherche Opérationnelle par la manière dont elle est mise en œuvre. Habituellement, les algorithmes doivent être adaptés aux caractéristiques du problème traité. Ce n'est pas le cas en programmation par contraintes où le problème traité est décrit à l'aide des outils disponibles dans la bibliothèque. L'exercice consiste alors à choisir avec attention quelles contraintes combiner pour exprimer correctement le problème, tout en tirant parti des avantages qu'elles offrent en terme d'efficacité. La pensée de Freuder, « *The user states the problem, the computer solves it.* », influe encore largement la manière dont un problème est traité avec cette méthode.

Problématiques

Aujourd'hui, le traitement d'un problème combinatoire s'inscrit dans une démarche itérative alternant une phase de modélisation avec une phase de résolution, l'analyse de chacune d'elles servant à l'amélioration de l'autre. L'un des intérêts de la programmation par contraintes est de réunir au sein d'une même entité l'ensemble de ces outils d'aide à la mo-

délisation et à la résolution : le solveur de contraintes. Une grande liberté est donnée au modeleur pour décrire son problème : en manipulant différents types de variables (booléennes, entières, ensemblistes, réelles, etc.), en utilisant des contraintes plus ou moins expressives ou efficaces (contraintes en extension, en intension, globales, décomposées, etc.). De la même manière, de nombreux algorithmes sont disponibles pour résoudre le problème : en profondeur d'abord ou en largeur d'abord, à grand voisinage, avec gestion des conflits, avec génération de clauses, etc.

Le mode opératoire exige, premièrement, d'élaborer un modèle du problème qui soit correct, complet et cohérent. Pour cela, bien entendu, les connaissances et l'expérience du modeleur sont largement sollicitées, et n'ont de limites que la déclarativité du solveur. Dans cette première étape, l'utilisateur voudra vérifier et affiner son modèle, le solveur est alors employé en mode « boîte noire », sans préciser la manière dont l'espace de recherche doit être parcouru. Les solveurs actuels proposent généralement des stratégies de branchement génériques, comme par exemple *Impact-based Search* [96], *Dow/Wdeg* [20] ou plus récemment, *Activity-based Search* [76], une adaptation de l'heuristique VSIDS [78] au CSP. Dès lors, suivant la typologie du problème, les attentes peuvent différer, et influenceront le choix de la techniques à appliquer. Pour un problème de satisfaction, s'agit-il plutôt de trouver une solution ou, au contraire, d'en prouver l'absence ? D'ordinaire, pour un problème d'optimisation, on se satisfera, dans un premier temps, d'obtenir une solution de « bonne qualité » en un « temps raisonnable » plutôt que d'en prouver l'optimalité au détriment du temps de calcul. Alors, le choix se portera, classiquement, vers une technique de Recherche Locale, sous la forme d'une Recherche à Voisinage Large, par exemple. Les stratégies de branchement génériques peuvent toujours être utilisées, mais il devient inévitable de s'attarder sur la conception de voisinages adaptés au problème traité. En effet, la description d'une heuristique de voisinage efficace repose généralement sur des connaissances métier, intrinsèquement liées à la classe de problèmes modélisés. Ces considérations d'ordre technique, dont la finalité est l'amélioration du processus de résolution, perturbent l'étape de modélisation. Nous sommes persuadés que la programmation par contraintes gagne en qualité et en visibilité en proposant des techniques génériques efficaces et accessibles au plus grand nombre.

Dans un second temps, et pour certaines applications pratiques, le modeleur peut vouloir maîtriser la combinatoire du problème théorique sous-jacent, ce qui passe par un contrôle accru du mécanisme de résolution. Ce contrôle est possible soit dans l'exploration de l'espace de recherche, soit dans l'interaction entre contraintes. Les informations relatives au modèle déclaré doivent pouvoir être exploitées afin de perfectionner le mécanisme de résolution. Les solveurs modernes offrent un niveau d'abstraction permettant de contrôler finement l'exploration de l'espace de recherche. Des systèmes avancés, comme les *Ilog Goals* [49], ou des langages dédiés, comme les *search combinators* [102], fournissent les éléments nécessaire au pilotage de la recherche. À défaut, la possibilité d'implémenter sa propre stratégie est communément offerte à l'utilisateur, le périmètre est alors clairement défini, limité et imposé par le langage de programmation. En revanche, le contrôle de l'ordonnancement des contraintes à propager reste, de manière générale, inexploité dans les solveurs actuels, principalement pour des raisons d'efficacité mais aussi, de part la nature des modifications à apporter à l'algorithme l'orchestrant. Dans ce contexte, une remise en cause de l'architecture existante des solveurs n'est pas suffisante, et la devise « diviser pour régner » chère au génie logiciel atteint ses limites. En effet, les solveurs de contraintes sont des applications complexes tant au niveau algorithmique qu'au niveau des structures de données. Trouver une bonne décomposition permettant à l'utilisateur d'agir localement pour spécifier un comportement particulier est souvent un casse-tête. Se situant en son cœur, les algorithmes de propagation sont généralement réalisés en amont du développement du solveur de contraintes. Leurs conceptions

influencent largement le reste du solveur, rendant certains choix quasiment définitifs. C'est pourquoi il est rare d'avoir une alternative à l'algorithme implémenté, ni même d'être en capacité de modifier celui en place. C'est là une limite à la déclarativité des solveurs de contraintes.

Contributions

Dans cette thèse, nous nous intéressons à proposer des solutions pour concevoir des solveurs « boîte grise » qui accompagnent un utilisateur depuis l'apprentissage jusqu'à l'expertise. Ces boites à outils doivent donc permettre de répondre aux besoins de simplification d'un néophyte, proposant une offre d'heuristiques de voisinage génériques s'adaptant à la nature du problème, et attentes de personnalisation d'un modeleur expert, en exposant certains mécanismes internes des solveurs qui pourront être composés *intelligemment* via un langage dédié à l'ordonnancement de la propagation.

En conséquence, l'objectif de cette thèse est double. D'une part, l'étude de la contribution d'un système à base d'explications pour l'amélioration d'une recherche locale sera menée. Il sera proposé diverses heuristiques permettant de s'affranchir de la conception de voisinages spécifiques au problème traité, répondant ainsi au besoin de concentrer d'abord la réflexion et les efforts sur la phase de modélisation. Ces heuristiques devront montrer que combiner recherche locale et explications permet de résoudre efficacement des problèmes d'optimisation. D'autre part, la spécification du comportement d'un moteur de propagation de contraintes sera considérée. Il sera proposé de faire un état des lieux des algorithmes de propagation pour les solveurs de contraintes, et d'exploiter opérationnellement ces informations à travers un langage dédié qui permette une personnalisation de la propagation au sein d'un solveur, en offrant des structures d'implémentation et en définissant des points de contrôle dans le solveur. Ce langage offrira des concepts de haut niveau permettant à l'utilisateur d'ignorer les détails de mise en œuvre du solveur, tout en conservant un bon niveau de flexibilité et certaines garanties. Il permettra l'expression d'une propagation spécifique à la structure interne de chaque problème, tout en conservant les mécanismes classiques connus.

La mise en œuvre et les expérimentations seront centrées autour de la nouvelle version de Choco ([90, 91]), le solveur de contraintes développé à l'École des mines de Nantes depuis 2003 par les membres de l'équipe TASC.

Organisation de la thèse

Le contexte de ce travail est introduit dans la partie I. Nous y présentons les notations et principes usuels de la programmation par contraintes. En particulier, nous nous attardons sur l'algorithme de propagation des contraintes mis en pratique dans les solveurs de contraintes, ainsi que les deux principales familles de moteurs. Nous décrirons aussi les différentes manières de procéder au parcours de l'espace de recherche. Nous détaillerons plus spécialement la recherche à voisinage large, et les algorithmes de backtrack intelligents.

Puis, dans la partie II, nous nous concentrons sur la finalité « boîte noire » des solveurs de contraintes, et décrivons comment la structure interne d'un problème, captée par les explications, peut être automatiquement mise à profit afin de construire des voisins au profit d'une recherche à voisinage large. Après avoir précisément définit la problématique autour des voisinages génériques (chapitre 3), nous décrivons, dans le chapitre 4, deux heuristiques qui exploitent singulièrement les explications pour construire de tels voisinages pour la re-

cherche à voisinage large. Ensuite, dans le chapitre 5, nous évaluons nos propositions, et diverses combinaisons, sur un ensemble varié de problèmes extraits de la librairie MiniZinc et comparerons les résultats avec ceux obtenus à l'aide des voisinages génériques standards. Ces travaux ouvrent de nouvelles pistes concernant l'usage des explications dans les solveurs de contraintes, elles sont exposées dans le chapitre 6.

Habituellement, les solveurs de contraintes offrent en plus la possibilité de maîtriser leurs fonctionnements internes. C'est pourquoi nous nous focalisons sur le caractère « boîte blanche »des solveurs dans la partie III, et abordons une manière de prototyper facilement des schémas de propagation des contraintes en exploitant la structure interne des problèmes traités. Nous formalisons, tout d'abord, dans le chapitre 7, les besoins et attentes des algorithmes de propagation. Puis, dans le chapitre 8, nous proposons un langage dédié à la description de moteur de propagation, et en listons certaines de ces propriétés et garanties. Nous discutons également des limites de notre proposition. Ensuite, nous évaluons notre langage de deux façons : premièrement, nous évaluons son impact sur le processus « normal » de résolution, en mesurant le surcoût induit par l'analyse syntactique et l'interprétation des expressions. Deuxièmement, nous montrons combien il est facile à utiliser pour rapidement concevoir des prototypes de moteurs de propagation corrects et variés. Nous présentons quelques remarques sur nos travaux et des propositions d'amélioration, notamment concernant le protocole expérimental (chapitre 10).

Enfin, dans la partie IV, nous concluons ce manuscrit en listant les contributions et en définissant des pistes pour des études futures. Nous évoquons également l'apport de cette thèse à l'amélioration du solveur Choco.

I

Contexte de l’étude

2

La Programmation par Contraintes

« *What is the role of CP solver developers in all this ?* »
(Håkan Kjellerstrand, 2013, CP Solvers – CP'13)

Sommaire

Dans ce chapitre, nous introduisons les notions nécessaires à la présentation des contributions (Partie II et Partie III). Dans la Section 2.1, nous empruntons les notions et notations des *Problèmes de Satisfaction de Contraintes* aux articles de Christian Bessière [8], d'une part, et Christian Schulte et Mats Carlsson [104], d'autre part, tous deux extraits du *Handbook of Constraint Programming* [99]. Nous nous intéressons aux algorithmes de propagation des contraintes (Section 2.2). Nous présentons les moteurs de propagation les plus communément communément utilisés en *Programmation Par Contraintes* (Section 2.2.2) ainsi que des améliorations successives de ces moteurs. Ensuite, nous présentons les algorithmes de parcours de l'espace de recherche (Section 2.3), des versions complètes arborescentes aux incomplètes basées sur la recherche locale, en passant par celles dites "intelligentes".

Nous invitons le lecteur désirant plus de détails à se référer aux documents cités précédemment.

2.1 Variables, Domaines et Contraintes

Les *Problèmes de Satisfaction de Contraintes* ou CSP (pour "*Constraint Satisfaction Problem*" en anglais), sont des problèmes mathématiques modélisés à l'aide de *variables*, de *domaines* et de *contraintes*. Les variables sont caractérisées par un domaine de définition, généralement fini. Les contraintes spécifient des combinaisons de valeurs auxquelles un sous-

ensemble de variables peut être affecté, établissant des relations entre les variables L'enjeu consiste à attribuer des valeurs aux variables qui satisfont toutes les contraintes, en utilisant les techniques de *Programmation Par Contraintes*, ou PPC.

Définition 2.1 (Contrainte [8]) *Une* contrainte c *est une relation définie sur un ensemble ordonné de* k *variables,* $X_c = (x_1, \ldots, x_k)$. c *est un sous-ensemble de* $\mathbb{Z}^{|X(c)|}$ *qui contient les combinaisons de valeurs (ou* tuples*)* $\tau \in \mathbb{Z}^{|X(c)|}$ *qui satisfont* c. $|X(c)|$ *est appelé l'*arité *de* c. *La fonction booléenne* $f_c : \tau \rightarrow \{true, false\}$ *teste si un tuple* τ *satisfait la contrainte.*

Une contrainte peut être spécifiée *en extension*, en énumérant les tuples autorisés, ou *en intension* en établissant un ensemble de règles portant sur les domaines des variables. Plus de détails concernant la qualification des contraintes seront donnés dans la Section 2.2.1. Un CSP est défini à l'aide d'un ensemble fini de contraintes et se modélise grâce à un *réseau de contraintes*.

Définition 2.2 (Réseau de contraintes [8]) *Un* réseau de contraintes *est composé de :*

- *une séquence finie* $X = (x_1, \ldots, x_n)$ *de* n *variables entières, et*
- *un domaine* $\mathcal{D}$ *associé à* X, $\mathcal{D} = D(x_1) \times \ldots \times D(x_n)$, *où* $D(x_i)$ *est un ensemble fini de valeurs entières auxquelles la variable* x_i *peut être affectée, et*
- *un ensemble de contraintes* $C = \{c_1, \ldots, c_m\}$, *portant sur les variables* $X(c_j) \in X$.

Étant donné un réseau N, ses variables, son domaine et ses contraintes seront référencés, respectivement, $X_N, \mathcal{D}_N$ et C_N. Étant donné une variable x_i et son domaine $D(x_i)$, $\underline{D}(x_i)$ référence la plus petite valeur de $D(x_i)$ et $\overline{D}(x_i)$ la plus grande. Les domaines sont équipés de la relation d'ordre suivante :

Définition 2.3 (Relation d'ordre [104]) *Étant donné deux domaines* $\mathcal{D}_1$ *et* $\mathcal{D}_2$, $\mathcal{D}_1$ *est* plus petit *que* $\mathcal{D}_2$, *noté* $\mathcal{D}_1 \sqsubseteq \mathcal{D}_2$, *si* $D_1(x) \subseteq D_2(x)$ *pour toutes variables* $x \in X$.

Exemple 2.1 (Plus petit) *Étant donné deux variables* x_1 *et* x_2, *si* $\mathcal{D}_1(x_1) = \{1\}$, $\mathcal{D}_1(x_2) = \{1, 2\}$ *et* $\mathcal{D}_2(x_1) = \{1, 2, 3\}$, $\mathcal{D}_2(x_2) = \{1, 2\}$ *alors* $\mathcal{D}_1 \sqsubseteq \mathcal{D}_2$.

Un domaine $\mathcal{D}_1$ est plus petit que (respectivement égal à) un domaine $\mathcal{D}_2$ par rapport au sous-ensemble de variables $Y \subseteq X$, noté $\mathcal{D}_1 \subseteq_Y \mathcal{D}_2$ (respectivement $\mathcal{D}_1 =_Y \mathcal{D}_2$) si $D_1(x) \subseteq_Y D_2(x)$ (respectivement $D_1(x) =_Y D_2(x)$) pour toutes variables $x \in Y$.

Résoudre un CSP consiste à affecter une valeur à chacune des variables jusqu'à ce qu'elles soient toutes *instanciées*.

Définition 2.4 (Instanciation [8]) *Étant donné un réseau* $N = (X, \mathcal{D}, C)$,

- *Pour un ensemble de variables* $Y = (x_1, \ldots, x_k) \subseteq X$, *une* instanciation $I[x_1, \ldots, x_k]$ *de* Y *est une affection des valeurs* $v_1, \ldots, v_k$ *aux variables* $x_1, \ldots, x_k$, *c.-à-d.* I *est un tuple pour* Y.
- *Une instanciation* I *de* Y *est* valide *si pour toutes variables* $x_i \in Y$, $I[x_i] \in D(x_i)$.
- *Une instanciation* I *de* Y *est* localement consistante *si et seulement si elle est valide et que pour toutes contraintes* $c \in C$ *telles que* $X(c) \subseteq Y$, $I[X(c)]$ *satisfait* c. *Si* I *n'est pas localement consistante, elle est* localement inconsistante.
- *Une* solution *au réseau* N *est une instanciation* I *de* X *qui est localement consistante.*

- *Une instanciation I de Y est* globalement consistante *(ou* consistante*) si elle peut être étendue à une solution.*

L'action d'instancier d'une variable x à une valeur v est notée $x \leftarrow v$, une variable x instanciée est notée $x^\star$ et sa valeur d'instanciation est notée $I[x]$. De manière analogue, l'action de supprimer une valeur v d'une variable x est notée $x \nleftarrow v$.

Exemple 2.2 (Instanciation) *Étant donné $N = (X, \mathcal{D}, C)$ où $X = (x_1, x_2, x_3, x_4)$, $D(x_i) = \{1, 2, 3, 4\}$ pour tout $i \in [\![1, 4]\!]$ et $C = \{c_1(x_1, x_2, x_3), c_2(x_1, x_2, x_3), c_3(x_2, x_4)\}$, telles que $c_1(x_1, x_2, x_3) \equiv (x_1 \neq x_2 \wedge x_1 \neq x_3 \wedge x_2 \neq x_3)$, $c_2(x_1, x_2, x_3) \equiv (x_1 \leq x_2 \leq x_3)$ et $c_3(x_2, x_4) \equiv (x_4 \geq 2.x_2)$:*

- *$I_1[x_1, x_2, x_4] = (1, 2, 7)$ n'est pas une instanciation valide de $Y_1 = (x_1, x_2, x_4)$;*
- *$I_2[x_1, x_2, x_4] = (1, 1, 3)$ est une instanciation localement consistante de $Y_2 = (x_1, x_2, x_4)$ puisque seule c_3 est couverte par Y_2 et elle est satisfaite par l'instanciation ;*
- *I_2 n'est pas globalement consistante puisqu'elle ne peut pas être étendue vers une solution, la contrainte c_1 ne pourra jamais être satisfaite par une extension de I_2.*

Il est possible d'étendre une instanciation soit en réduisant $\mathcal{D}_N$ (*p. ex.*, une variable est instanciée), soit en ajoutant de nouvelles contraintes à C_N (*p. ex.*, une nouvelle relation entre variables est définie). On dit alors que le réseau est *contracté*. La transformation du réseau N qui en résulte requiert de vérifier la consistance locale de N ainsi transformé. En effet, construire un réseau globalement consistant est exponentiel en temps, mais de plus, sa taille croit généralement de manière exponentielle en fonction de N [8]. Habituellement, on cherche à s'approcher par construction de la consistance globale à un coût plus raisonnable, en pratique dans une complexité temporelle et spatiale polynomiale. Cette approximation s'effectue au travers de la *propagation de contraintes*, mécanisme se situant au cœur du processus de résolution. Les opérations à exécuter pour propager les contraintes définissent l'*algorithme de propagation*.

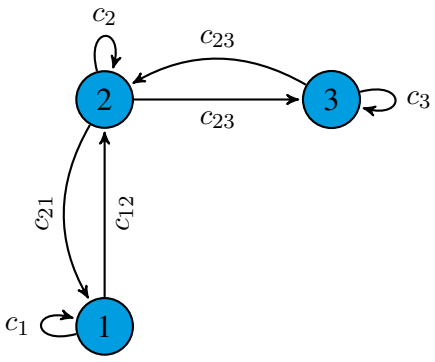
2.2 L'algorithme de propagation

Dans [69], Alan Mackworth présente l'algorithme nommé AC-3, où AC vaut pour *Arc Consistency* – Consistance d'Arc. Cet algorithme a été proposé pour un réseau de contraintes binaires : le CSP est composé de contraintes en extension d'arité 2 maximum. Les contraintes sont soit *unaires*, *c.-à-d.* impliquant une seule variable, soit *binaires*, *c.-à-d.* impliquant deux variables distinctes. Soit x_i une variable, nous supposons que :

- Il y a au plus une contrainte unaire impliquant x_i, nommée c_i, qui représente le domaine de x_i ;
- Soit x_j une autre variable ($i \neq j$), il y a au plus une contrainte qui implique les deux variables, nommée c_{ij}.

Généralement, un réseau N est représenté par un graphe orienté G_N afin de décrire les algorithmes de consistance d'arc. Chaque nœud i de G_N représente une variable x_i de N. Une contrainte unaire c_i est représentée par une boucle sur le nœud i. Pour chaque contrainte c_{ij} impliquant les variables x_i et x_j, deux arcs orientés sont représentés : (x_i, c_{ij}) et (x_j, c_{ji}). c_{ij} et c_{ji} représentent la même contrainte, seuls les paramètres de la fonction booléenne associée sont intervertis : $f_{c_{ij}}(v_i, v_j) = f_{c_{ji}}(v_j, v_i), v_i \in D(x_i)$ et $v_j \in D(x_j)$.

FIGURE 2.1 – Représentation d'un CSP sous la forme d'un graphe.



Exemple 2.3 (CSP sous forme de graphe) *Étant donné le CSP suivant composé de :*

- *trois variables* $X = (x_1, x_2, x_3)$,
- *de leurs domaines, définis par des contraintes unaires* (c_1, c_2, c_3), *où :*
  - $c_1(x_1) \equiv (x_1 \in \{1, 2, 3\})$,
  - $c_2(x_2) \equiv (x_2 \in \{1, 2, 3\})$ *et*
  - $c_3(x_3) \equiv (x_3 \in \{1, 2, 3\})$,
- *et des contraintes* $C = (c_{12}, c_{23})$, *où*
  - $c_A(x_1, x_2) \equiv (x_1 + x_2 = 4)$ *et*
  - $c_B(x_2, x_3) \equiv (x_2 < x_3)$,

sa représentation sous la forme d'un graphe orienté est donné dans la Figure 2.1. Les arcs c_{12} *et* c_{21} *représentent la contrainte* c_A*, les arcs* c_{23} *et* c_{32} *représentent la contrainte* c_B.

Alan Mackworth définit également la notion de consistance d'arc.

Définition 2.5 (Consistance d'arc) *L'arc* (x_i, c_{ij}) *est* arc-consistant *si et seulement si pour toute valeur* $v \in D(x_i)$ *il existe une valeur* $w \in D(x_j)$ *telle que* $f_{c_{ij}}(v, w)$ *est vraie. On dit alors que* v *a un* support *dans* c_{ij}.

Un CSP est dit arc-consistant si, et seulement si, chaque arc est arc-consistant. Si la valeur $v \in D(x_i)$ n'a pas de support dans la contrainte c_{ij}, alors v doit être supprimée de $D(x_i)$. Lorsque cette vérification a été faite pour toutes les valeurs de $D(x_i)$, alors l'arc (x_i, c_{ij}) est consistant. Cela ne vaut pas automatiquement pour l'arc (x_j, c_{ji}). Vérifier tous les arcs une fois n'est pas suffisant pour garantir que le réseau associé est localement consistant. En effet, chaque retrait de valeur du domaine d'une variable peut remettre en cause la consistance locale des autres contraintes de cette variable. Les opérations de suppression des valeurs du domaine de la variable x_i sans support pour une contrainte c_{ij} sont décrites dans l'Algorithme 1. Cet algorithme renvoie un booléen indiquant si au moins une valeur a été supprimée.

L'Algorithme 2 décrit comment atteindre la consistance d'arc sur un CSP. Tous les arcs du graphe sont placés dans une file Q (ligne 2), on dira par la suite qu'ils sont *planifiés*. Puis, tant que cette file n'est pas vide (ligne 3), un arc est sélectionné et supprimé de Q pour être révisé (lignes 4 et 5). Si au moins une valeur a été supprimée du domaine de la variable x_i, alors il est nécessaire de vérifier si toutes les contraintes impliquant x_i sont toujours

Algorithme 1 Algorithme de suppression des valeurs sans support.

```
1:  procedure REVISE(x_i, c_ij)
2:      m ←false
3:      for v ∈ D(x_i) do
4:          if ∀w ∈ D(x_j), f_{c_ij}(v, w) =false then
5:              D(x_i) ← D(x_i) \ {v}
6:              m ←true
7:          end if
8:      end for
9:      return m
10: end procedure
```

Algorithme 2 Algorithme AC-3.

```
1:  procedure PROPAGATE_AC-3(X, D, C)
2:      Q ← {(x_i, c)|c ∈ C, x_i ∈ X(c)}
3:      while Q ≠ ∅ do
4:          (x_i, c) ←DEQUEUE(Q)
5:          if REVISE(x_i, c) then
6:              if D(x_i) = ∅ then
7:                  return false
8:              end if
9:              Q ← Q ∪ {x_j, c'|c' ∈ C ∧ c' ≠ c ∧ x_i, x_j ∈ X(c') ∧ j ≠ i
10:         end if
11:     end while
12:     return true
13: end procedure
```

localement consistantes en ajoutant les arcs concernés dans Q (ligne 9). Lorsque la file est vide, tous les événements ont été propagés sans détecter d'inconsistance, le CSP est alors localement consistant, la méthode retourne `true`. Si, au contraire, une erreur est rencontrée, *p. ex.*, le domaine d'une variable a été vidé, la propagation est stoppée, et la méthode retourne `false`.

La version présentée ici est *orientée arc*, puisque la file Q stocke et retourne des arcs à réviser. McGregor [75] propose non plus de gérer des arcs mais les variables modifiées, qualifiant alors l'algorithme d'*orienté variable*. À chaque modification d'une variable x_i, celle-ci est ajoutée dans Q. Lorsqu'un variable est sélectionnée, un parcours de ses arcs est effectué pour qu'ils soient révisés. Bien que cette implémentation soit la plus répandue en pratique, la version arc est plus souvent utilisée dans la littérature.

Propriété 2.1 (Complexité AC-3) *AC-3 permet d'atteindre la consistance d'arc avec une complexité temporelle de $O(md^3)$ et spatiale de $O(m)$, où m est le nombre de contraintes et d est la taille du plus grand domaine parmi ceux composant $\mathcal{D}$.*

Lorsqu'une valeur est supprimée du domaine d'une variable, il faut, pour une contrainte donnée, parcourir ses tuples pour en vérifier la validité (Algorithme 1, ligne 4). La complexité temporelle de la méthode REVISE (Algorithme 1) est sous optimale : soit la valeur a un support, quel qu'il soit, et il n'est pas nécessaire de parcourir toutes les combinaisons. Soit la valeur n'a plus de support, et il faut en prouver l'absence. Cette observation est prise en compte dans l'algorithme AC-4 proposé par Mohr et Henderson [77] : réduire l'effort de

recherche d'absence de support pour une valeur, en associant un compteur à chaque triplet (x_i, v_i, c_{ij}). Ce compteur dénombre les supports de la valeur $v_i \in D(x_i)$ pour la contrainte c_{ij}. Cette version de l'algorithme nécessite une phase d'initialisation au cours de laquelle le nombre de supports pour chaque triplet est calculé. Ce compteur est ensuite maintenu incrémentalement, à mesure que des valeurs sont retirées. Quand le compteur du triplet (x_i, v_i, c_{ij}) a pour valeur zéro, cela signifie qu'il n'existe plus de support pour la valeur v_i de la variable x_i pour la contrainte c_{ij}. Cette valeur est donc supprimée du domaine de x_i et révision des arcs de la variable est nécessaire.

Propriété 2.2 (Complexité AC-4) *AC-4 permet d'atteindre la consistance d'arc avec une complexité temporelle de $O(md^2)$ et spatiale de $O(md^2)$.*

AC-4 est le premier algorithme qualifié de *grain fin* puisqu'il base la propagation sur les valeurs plutôt que sur les arcs (AC-3 est un algorithme *gros grain*). Cependant, bien que AC-4 soit optimal en temps, il souffre d'une importante occupation mémoire due aux compteurs. De plus, il nécessite une phase d'initialisation. Une nouvelle amélioration permet de réduire l'espace mémoire tout en maintenant la complexité temporelle. Il s'agit de garantir qu'il existe au moins un support pour une valeur, sans les dénombrer. Bessière et Cordier [7, 9] introduisent l'algorithme AC-6 : l'étape d'initialisation consiste à trouver le *plus petit support* (au sens lexicographique) pour chaque valeur v de la variable x_i pour chaque contrainte c_{ij}. Lorsque le plus petit support de v n'est plus valide, AC-6 impose de chercher le prochain. Cette recherche s'effectue en parcourant les combinaisons en commençant à $v + 1$; les supports inférieurs n'étant naturellement plus valides.

Propriété 2.3 (Complexité AC-6) *AC-6 permet d'atteindre la consistance d'arc avec une complexité temporelle de $O(md^2)$ et spatiale de $O(md)$.*

Le principe de mémorisation d'un support pour un triplet est adaptable à l'algorithme gros grains AC-3. Il suffit pour cela de modifier la méthode REVISE : un pointeur vers le support de plus petite valeur est maintenu pour chacune des valeurs des variables sur chaque contrainte. On obtient alors l'algorithme AC-2001 [14, 15, 118].

Propriété 2.4 (Complexité AC-2001) *AC-2001 permet d'atteindre la consistance d'arc avec une complexité temporelle de $O(md^2)$ et spatiale de $O(md)$.*

D'autres extensions de l'algorithme d'arc-consistance ont été proposées (AC-7 [10, 11, 13], AC-8 [26], AC-3.2, AC-3.3 [66]). Dans [97], Régin a proposé un algorithme d'arc consistance générique, configurable et adaptatif, appelé CAC, permettant de combiner les techniques des algorithmes d'AC existantes.

D'autres niveaux de consistance, plus forts, ont également été étudiés, comme la consistance de domaine [31, 16], ou plus faibles, comme la consistance aux bornes ou consistance d'intervalles[1]. On parle d'*arc consistance généralisé* lorsque le réseau de contraintes est composé de contraintes d'arité non fixée. Van Hentenrick et autres [48] ont généralisé les algorithmes AC-3 et AC-4 en AC-5, pour gérer naturellement des contraintes hétérogènes.

2.2.1 Hétérogénéités du réseau de contraintes

Dans ces algorithmes d'arc-consistance présentés dans la Section 2.2, l'algorithme de suppression des valeurs sans supports (Algorithme 1) est commun à toutes les contraintes, caractérisant l'homogénéité des contraintes. Toutefois, la modélisation autorise l'utilisation de contraintes hétérogènes, dont l'implémentation de la méthode REVISE peut être spécifiée.

[1] Des travaux récents [4] étudient l'adaptation de la consistance au problème durant pendant la résolution.

Comme nous l'avons indiqué brièvement dans la Section 2.1, une contrainte peut être définie *en extension* ou *en intension*.

En extension

Une contrainte est spécifiée *en extension* en listant les tuples qui la satisfont. Le nombre de tuples représenté en mémoire pour les contraintes est borné par n^d où n est la plus grande arité des contraintes de C et d est la taille du plus grand domaine parmi ceux composant $\mathcal{D}$. Définir les contraintes en extension n'est, de ce fait, pas toujours réalisable.

Exemple 2.4 (En extension) *Étant données deux variable x_1 et x_2 et leurs domaines $D(x_1) = \{1, 2, 3, 4\}$ et $D(x_2) = \{1, 2, 3, 4\}$, la contrainte "x_1* est strictement inférieure à *x_2" s'exprime en extension $c(x_1, x_2) = \{(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)\}$, en limitant les valeurs pour x_1 à 1, 2, et 3, et x_2 à 2, 3 et 4.*

Il peut être possible de les définir *en intension*.

En intension

Une contrainte est spécifiée *en intension* en établissant un ensemble de règles portant sur les domaines des variables. Les contraintes en intension, en comparaison avec celles en extension, fournissent leur propre version de la méthode REVISE, tirant profit de la sémantique de la contrainte et réduisant potentiellement sa complexité algorithmique spatiale et/ou temporelle.

Exemple 2.5 (En intension) *Étant données deux variable x_1 et x_2 et leurs domaines $D(x_1) = \{1, 2, 3, 4\}$ et $D(x_2) = \{1, 2, 3, 4\}$, la contrainte "x_1* est strictement inférieure à *x_2" s'exprime en intension grâce aux deux règles suivantes :*

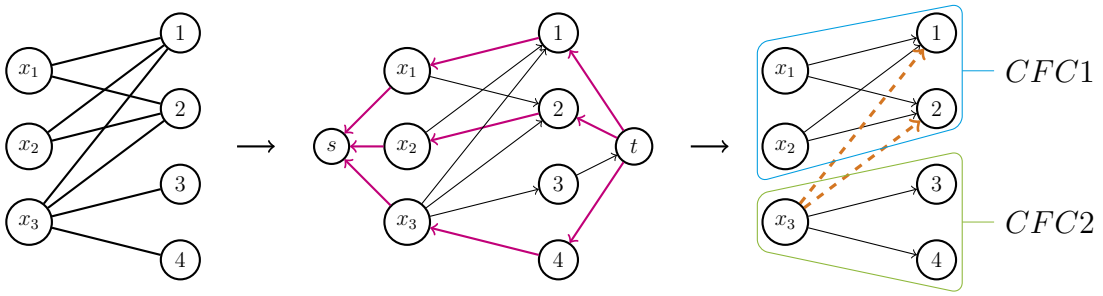
- $\overline{D}(x_1) \leftarrow \overline{D}(x_2) - 1$
- $\underline{D}(x_2) \leftarrow \underline{D}(x_1) + 1$

Lorsqu'une contrainte en intension est spécifiée sur un nombre indéterminé de variables, on parle alors de contrainte *globale*. Bessière et Van Hentenrick [12] ont proposé une définition de contrainte globale, en introduisant plusieurs notions de globalité :

- Une contrainte est *sémantiquement globale* s'il n'existe pas de décomposition de cette contrainte à l'aide de contraintes de plus faible arité,
- Une contrainte est *opérationnellement globale* s'il n'existe pas de décomposition sémantiquement globale qui garantisse le même niveau de consistance (*c.-à-d.* qui permette de filtrer autant de valeurs inconsistantes),
- Une contrainte est *algorithmiquement globale* s'il n'existe pas de décomposition opérationnellement globale de même complexité algorithmique temporelle et spatiale.

La contrainte $c_A \equiv \texttt{AllDifferent}(x_1, \ldots, x_n)$ est sans doute l'une des contraintes globales les plus connues et la plus simple à décrire. Elle impose que les n variables qu'elle implique prennent chacune une valeur distincte dans une solution. Cette contrainte n'est pas sémantiquement globale, il est possible de l'exprimer à l'aide de $\frac{n \times (n-1)}{2}$ contraintes binaires "$\neq$" chacune d'entre elles exprimant "x_i *est différent de* x_j" (avec $i \neq j$). Deux implémentations de cette contrainte existent [68, 100], chacune étant opérationnellement globale. La

FIGURE 2.2 – Représentation de la contrainte `AllDifferent`(x_1, x_2, x_3) sous la forme d'un graphe biparti.



première implémentation [100], en $O(k\sqrt{n})$ où k est la somme de la taille des domaines, assure la *consistance d'arc généralisée*, *c.-à-d.* qu'il existe un support pour toute valeur $v \in D(x_i)$, quelle que soit $x_i \in X(c_A)$[2]. Sans entrer dans les détails d'implémentation, l'exemple 2.6 illustre un cas d'utilisation de la contrainte `AllDifferent` dans lequel il n'existe pas de décomposition assurant la consistance d'arc généralisée.

Exemple 2.6 (Globale : `AllDifferent`) *Etant données trois variables x_1, x_2 et x_3, leur domaine $D(x_1) = \{1, 2\}$, $D(x_2) = \{1, 2\}$ et $D(x_3) = \{1, 2, 3, 4\}$, les valeurs 1 et 2 sont supprimées de $D(x_3)$ par la contrainte `AllDifferent`(x_1, x_2, x_3). Le raisonnement est schématiquement représenté dans la Figure 2.2. Un graphe biparti est construit à partir des variables et des valeurs. Puis, un flot maximal est calculé en affectant une capacité de 1 à chaque arc. Enfin, les composantes fortement connexes sont calculées. Les arcs reliant deux composantes fortement connexes peuvent être supprimés (en pointillé sur la figure).*

La seconde implémentation [68], en $O(n \log n)$, assure la *consistance aux bornes*, *c.-à-d.* qu'il existe un support pour les bornes $\underline{D}(x)$ et $\overline{D}(x)$, quelle que soit $x_i \in X(c_A)$. Ces deux implémentations de la contrainte `AllDifferent`, bien que sémantiquement équivalentes, ne sont pas comparables en terme de *puissance* du filtrage associé (les deux algorithmes n'atteignent pas la même consistance), ni de *coût* (temps de calcul). Cependant, ces deux versions peuvent tout à fait co-exister au sein d'un solveur de programmation par contraintes, laissant à l'utilisateur la liberté de choisir l'algorithme de filtrage le plus adapté au problème traité. Un catalogue exhaustif des contraintes globales est maintenu par Beldiceanu et al. [6].

Propagateurs

Jusqu'à maintenant nous n'avons pas précisé la manière dont les contraintes supprimaient, ou filtraient, des valeurs du domaine des variables. En pratique, une contrainte est équipée d'un ou plusieurs algorithmes de filtrage, nommés *propagateurs*.

Définition 2.6 (Propagateur [104]) *Étant donné un réseau $N = (X_N, \mathcal{D}_N, C_N)$, un* propagateur *$p$ pour $c \in C_N$ est une fonction décroissante et monotone de $\mathcal{D}_N$ dans $\mathcal{D}_N$.*

L'ensemble des propagateurs d'un réseau N est noté P_N. Un propagateur réduit les domaines de $X(c)$ en filtrant les valeurs localement inconsistantes. La propriété de décroissance, *c.-à-d.* $p(\mathcal{D}) \subseteq \mathcal{D}$, garantit que la propagation ne fait que retirer des valeurs des domaines. La

[2]Pour ce faire, un graphe biparti est construit, dont les variables, d'une part, et les valeurs, d'autre part, constituent les sommets et les relations d'affection définissent les arêtes. Puis, un flot maximal est calculé en affectant une capacité de 1 à chaque arc. Enfin, les composantes fortement connexes sont calculées sur le graphe ainsi obtenu. Nous invitons le lecteur qui souhaite avoir plus d'information à propos des notions de *flot maximal* et *composantes fortement connexes* et la descriptions des algorithmes permettant de les calculer à consulter l'ouvrage intitulé "Graphs and Algorithms" [44].

propriété de monotonie, *c.-à-d.* $\mathcal{D}_1 \subseteq \mathcal{D}_2$ implique $p(\mathcal{D}_1) \subseteq p(\mathcal{D}_2)$, assure que l'application d'un propagateur sur un domaine plus petit produit également un domaine plus petit. Un propagateur doit également être *correct* pour une contrainte c, *c.-à-d.* il ne supprime pas de valeurs localement consistantes : $\{d \in \mathcal{D}\} \cap c = \{d \in p(\mathcal{D})\} \cap c$. Il est également pertinent de rendre les propagateurs *idempotents*, *c.-à-d.* $p(\mathcal{D}) = p(p(\mathcal{D}))$. Alors, l'ordre dans lequel ils sont exécutés n'a plus d'importance sur la qualité du filtrage.

Sous ces conditions, lorsqu'un propagateur a filtré toutes les valeurs localement inconsistantes de $X(c)$, il est à un *point fixe local*.

Définition 2.7 (Point fixe local) *Étant donné un réseau $N = (X, \mathcal{D}, C)$, un propagateur $p \in P_N$ est à un* point fixe local *si $p(\mathcal{D}) = \mathcal{D}$.*

Propriété 2.5 (Point fixe) *Étant donné un réseau $N = (X, \mathcal{D}, C)$ et P l'ensemble fini des propagateurs de N, si tous les propagateurs $p \in P$ sont à un point fixe local, alors le CSP est au point fixe.*

La propriété d'idempotence rend le point fixe unique quelque soit l'ordre dans lequel les propagateurs sont exécutés.

L'algorithme 3 présente un *moteur de propagation* qui atteint le point fixe pour un réseau de contraintes N donné. PROPAGATORS(C) retourne tous les propagateurs du réseau, $X(p)$ retourne les variables du propagateurs p.

Algorithme 3 Moteur de propagation basique.

```
1:  procedure PROPAGATE(X, D, C)
2:      P ←PROPAGATORS(C)
3:      Q ← P
4:      while Q ≠ ∅ do
5:          q ←SELECT(Q)
6:          REMOVE(q, Q)
7:          D' ← q(D)
8:          if D ≠ D' then
9:              M ← {x ∈ X | D(x) ≠ D'(x)}
10:             Q ← Q ∪ {p | p ∈ P, X(p) ∩ M}
11:             D ← D'
12:         end if
13:     end while
14: end procedure
```

Tant qu'il reste des propagateurs dans la file Q (Ligne 4), l'un d'entre eux est sélectionné (Ligne 5) pour être exécuté (Ligne 7). Si le domaine est réduit par propagation (Ligne 10), *c.-à-d.* si au moins une variable a été modifiée, l'ensemble des propagateurs concernés (qui impliquent l'une des variables modifiées) sont planifiés, ajoutés dans Q (Ligne 10). Notons que cet algorithme de propagation est orienté propagateurs.

Naturellement, cet algorithme fait l'objet d'adaptations pour sa mise en pratique. Par exemple, il est fréquent de ne pas considérer un propagateur comme une fonction de $\mathcal{D}_N$ dans $\mathcal{D}_N$, mais de restreindre les ensembles de départ et d'arrivée de la fonction aux domaines que le propagateur va consulter et/ou modifier. D'autre part, il est envisageable de définir une contrainte comme une combinaison de propagateurs. Ils se distinguent par la qualité de leur filtrage, leur complexité spatiale ou temporelle. Schulte et Carlsson [104]

estiment d'ailleurs que la consistance atteinte par un propagateur n'importe plus véritablement, autrement que pour comparer les propagateurs entre eux, puisqu'ils attendent d'un propagateur qu'il filtre bien ("*As far as achieving a good propagation is concerned, it does not matter wether a propagator set corresponds to a predefined consistency level*"). Nous allons maintenant décrire les éléments techniques pouvant varier d'une implémentation à une autre.

2.2.2 Dans un solveur de contraintes

Nous avons présenté les algorithmes de propagation sous trois orientations différentes : les algorithmes de propagation orientés arcs (Section 2.2), orientés variables (Section 2.2) ou orientés contraintes/propagateurs (Section 2.2.1). La manière dont Q est *orienté* peut avoir une incidence sur la manière dont les propagateurs doivent être implémentés. L'orientation désigne le type des objets qui vont être planifiés dans la file Q. Comme nous l'avons vu, les alternatives aux arcs sont (a) de planifier des variables [75], (b) de planifier des propagateurs [24]. Dans le cas d'un moteur de propagation *orienté variables* (a), lorsqu'une variable est retirée de l'ensemble Q, une boucle exécute ses propagateurs un à un. Dans le cas d'un moteur de propagation *orienté propagateurs* (b), lorsqu'une variable est modifiée, une boucle planifie ses propagateurs un à un. La complexité reste inchangée ; cependant, le nombre maximum d'objets ajoutés dans Q peut varier. D'autre part, l'ordre d'exécution des propagateurs, également, varie également d'une orientation à l'autre.

L'intérêt d'orienter un moteur autour des variables, au-delà de moins solliciter la file Q, est de pouvoir réagir facilement aux événements au fur et à mesure de leur génération. En contre-partie, cela peut devenir une faiblesse pour certaines contraintes globales. Par exemple, le propagateur associé à une contrainte `AllDifferent`, tel qu'il est décrit dans [100], nécessite, dans un premier temps, de mettre à jour une structure interne (un graphe biparti variable-valeur), puis dans un second temps, d'en extraire des informations permettant de filtrer des valeurs du domaine des variables. Il est tout à fait pertinent de maintenir incrémentalement le graphe biparti au fur et à mesure que les variables sont modifiées, mais de ne déclencher l'étape de filtrage proprement que ponctuellement. Pour adresser cette difficulté, une file de propagation "gros grains" est prévue [62, 37]. Une contrainte réagit alors aux événements fins, par exemple en maintenant incrémentalement la structure interne de la contrainte, mais retarde l'exécution de l'algorithme "lourd" dans une file annexe. Lorsque tous les événements fins ont été traités, un événement lourd est traité. Cette mutualisation des événements permet de déclencher l'algorithme moins souvent, d'économiser sur le temps d'exécution global sans réduire le niveau de consistance atteint par celui-ci. Les événements lourds peuvent être traités de manière plus précise en associant une priorité à chaque propagateur, pouvant qualifier leur complexité. Il convient alors de traiter en premier lieu les propagateurs les plus rapides [62]. La file de propagations peut être moins sollicitée en associant des *watched literals* [38] aux propagateurs. Le principe des watched literals vient de la communauté SAT [78] : il s'agit de sélectionner deux littéraux non instanciés dans chaque clause. À partir de ces pointeurs, il est possible de faire d'appliquer la règle suivante lorsqu'une valeur leur est attribuée : si un événement est généré sur l'une de ces variables, il est traité et l'algorithme de filtrage peut être exécuté. En revanche, s'il ne concerne aucune des deux variables, il est ignoré. Par exemple, la contrainte $\sum_{i=1}^{n} b_i \geq c$, où les variables b_i sont booléennes, se prête bien aux watched literals.

L'intérêt principal d'orienter un moteur autour des propagateurs est de permettre la mutualisation des événements par propagateur et donc d'en capitaliser les réveils. Cependant, le filtrage trivial peut alors être retardé. Par exemple, le propagateur associé à la contrainte *AllDifferent* [100] filtre des valeurs en se basant sur le graphe biparti variable-valeur. Un moteur de propagation orienté propagateurs réagit globalement aux modifications des variables subies depuis son dernier réveil. Le graphe biparti n'est alors pas maintenu incrémentalement. De plus, aucun filtrage spécifique à l'instanciation d'une des variables n'est appliqué. Il est pourtant pertinent de retirer cette valeur d'instanciation du domaine des autres variables impliquées, tel que cela est décrit dans [40], et ainsi, alimenter la file de propagation en événements rapidement propageables sur les contraintes rapides avant de déclencher le filtrage plus lourd. Les propagateurs à états [106] permettent de répondre à cette problématique. Naturellement, associer une priorité aux propagateurs permet de les exécuter en prenant en compte leur complexité, qui peut évoluer dynamiquement au cours de la résolution [105]. D'ailleurs, l'une des structures de propagation la plus efficace est décrite dans [106]. Une autre difficulté consiste à gérer les valeurs supprimées au fur et à mesure de la propagation. Grâce aux *advisors* [63], il est possible de simplifier la gestion de l'incrémentalité des propagateurs. Ces observateurs peuvent, par exemple, tenir les propagateurs informés des valeurs qui ont été supprimées des variables. Il également de définir des règles de planification des propagateurs. S'il est possible de définir des règles simples déterminant si un propagateur ne pourra rien déduire de l'état actuel des domaines des variables, on évite alors des exécutions coûteuses et stériles des propagateurs résultant de planifications inutiles.

D'autres techniques s'adaptent facilement quelle que soit l'orientation des moteurs. On peut citer par exemple la qualification des événements subis par les variables [62] (instanciation, modification de bornes, retrait de valeurs) ainsi que l'ensemble ordonné de ces événements [107]. Grâce à ces événements, il est possible de définir les événements dont dépendent les propagateurs [106], *c.-à-d.* ceux à partir desquels ils peuvent effectivement filtrer.

Ces techniques ont toutes le même objectif : réduire le nombre d'appels à la planification de propagateurs et rendre la phase de propagation plus efficace. En parallèle de ces travaux sur les conditions de planification, l'*ordre de révision* des objets à propager a été étudié. L'ordre de révision définit comment un élément est sélectionné et retiré de l'ensemble Q. Il est généralement admis qu'une file, qui choisit l'élément le plus vieux d'abord, est un bon choix pour l'ensemble Q car il traite tous les éléments de manière équitable. Cependant, des alternatives existent. D'une part, [19] propose la sélection dynamique d'un élément en se basant sur un critère tel que la taille du domaine de la variable ou l'arité d'un propagateur. D'autre part, [106] ont proposé un moteur orienté propagateurs avec sept niveaux de priorités. Un propagateur est planifié dans une file en fonction de sa priorité dynamique[3]. Enfin, de récents progrès ont été faits concernant l'accès à l'algorithme de propagation en introduisant la notion de groupe de propagateurs [64]. Ces groupes autorisent la description de schéma de propagation spécifique à un ensemble de propagateurs. Un propagateur additionnel est crée pour "piloter" le groupe et le représenter dans le moteur de propagation. Néanmoins, l'implantation des groupes reste à la charge de l'utilisateur. Le Tableau 2.1 récapitule les techniques présentées ci-dessus.

Bien entendu, l'algorithme de propagation s'insère dans un algorithme qui pilote l'exploration de l'espace de recherche. Nous décrivons plus spécialement cet algorithme dans la section suivante.

[3] La priorité dynamique combine l'arité et la priorité pour fournir une évaluation fine du coût d'exécution du propagateur [106].

TABLE 2.1 – Tableau récapitulatif des techniques d'amélioration de l'algorithme de propagation.

Problématique	Réponses adressées
Réduire la planification, améliorer l'incrémentalité	Watched literals [38], advisors [63], raisonnement par événements [62, 107], dépendance des propagateurs [106]
Gérer les complexités	Queues de contraintes additionnelles [62, 37], propagateurs à priorités [62, 105], moteur orienté propagateurs à sept niveaux de priorités [106], propagateurs à états [106]
Réduire les exécutions	contrôle statistique et probabilités [110, 21, 18]
Revoir l'ordre de révision	Sélection dynamique [19, 115], moteur orienté propagateurs à sept niveaux de priorités [106], groupe de propagateurs [64]

2.3 L'algorithme de *backtrack* et ses améliorations

La résolution d'un CSP est généralement combinatoire : une énumération des instanciations possibles doit être faite pour en trouver si possible au moins une qui satisfait l'ensemble des contraintes, *c.-à-d.* qui soit valide. L'algorithme de base, appelé *generate and test*, génère des instanciations valides et teste leur consistance en parcourant les contraintes du réseau, n'est pas utilisable en pratique. En effet, le nombre de combinaisons possibles, définissant l'espace de recherche ER, est égal à :

$$ER = \prod_{i=1}^{n} |D(x_i)|$$

Si tous les domaines sont de même taille k, alors $ER = k^n$. Le nombre d'instanciations à générer augmente de manière exponentielle en fonction de $|X|$ et $|\mathcal{D}|$. Une première manière simple d'améliorer l'algorithme précédent consiste à faire contrôler, par les contraintes, les instanciations partielles au fur et à mesure de leur construction. Un processus constructif procède de la manière suivante : à chaque étape, une variable non instanciée est sélectionnée pour être instanciée à une valeur de son domaine. La manière dont les variables, et les valeurs d'instanciation, sont sélectionnées définit une *stratégie de branchement*. Cette stratégie construit itérativement un arbre de recherche où un nœud est une instanciation partielle, une branche est une extension d'instanciation partielle et une feuille est une instanciation totale.

Cette stratégie peut être définie de deux manières :

- *énumerative* : une variable est sélectionnée puis instanciée, tour à tour, à chacune des valeurs de son domaine. On parle alors d'un *branchement n-aire* puisque chaque nœud de l'arbre de recherche peut conduire à créer n nœuds fils. Une variable peut n'être associée qu'à un seul nœud ;

- *binaire* : une variable est sélectionnée et une valeur de son domaine est choisie. Tout d'abord, l'instanciation est appliquée, puis, si nécessaire, elle est interdite. Chaque nœud de l'arbre de recherche peut donc conduire à créer au maximum deux nœuds fils. Une variable peut être associée à plusieurs nœuds.

Il est possible de ne pas instancier la variable à la valeur, mais de découper le domaine en deux autour de cette variable, à l'aide des opérateurs $\leq, >, \geq$ et $<$. On parle alors de *domain splitting*.

La réduction d'un domaine d'une variable, représentée par le triplet variable-opérateur-valeur, est nommée *décision*, et peut être vue comme une nouvelle contrainte ajoutée dynamiquement au CSP. Par la suite, nous ne considérerons que les branchements binaires où une décision sera définie comme :

Définition 2.8 (Décision) *Une décision est un triplet $\delta = \langle x, o, v \rangle$, où x est une variable dont $|D(x)| > 1$, o un opérateur réfutable et $v \in D(x)$ une valeur. Par défaut, l'opérateur est $=$, il pourra être ignoré dans la déclaration d'une décision : $\delta = \langle x, v \rangle$.*

Définition 2.9 (Chemin de décisions) *Un* chemin de décisions *est un ensemble chronologiquement ordonné de décisions.*

Puisqu'elle contracte le réseau, l'application d'une décision doit être validée par la propagation. De la même manière, la *réfutation* d'une décision, *c.-à-d.* interdire qu'elle puisse être de nouveau appliquée dans le sous-arbre de recherche, doit être validée. Si le réseau contracté est inconsistant, l'algorithme remet en cause le choix de la valeur, ou de la variable, pour poursuivre la construction d'une solution. Ces remises en cause sont qualifiées de "retours arrières", ou *backtrack* en anglais, qui donne ainsi son nom à l'algorithme de recherche.

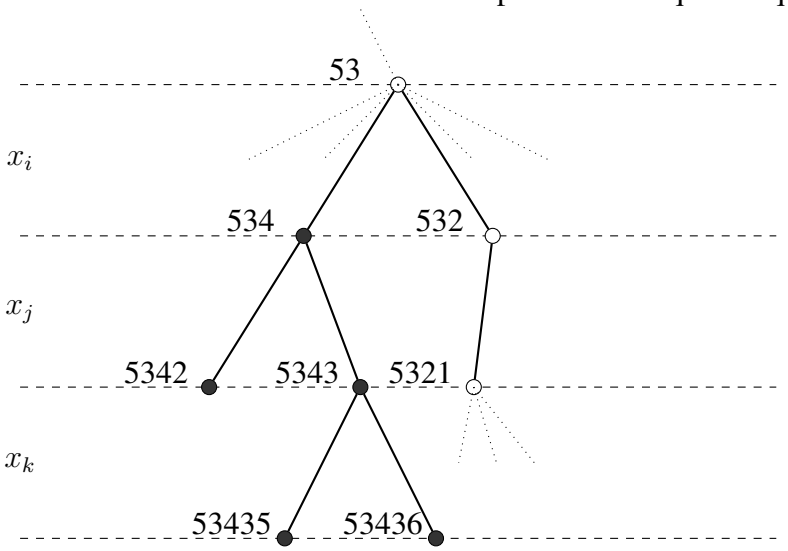
Exemple 2.7 (Arbre de recherche) *Un extrait d'un arbre de recherche, pour un CSP quelconque, est donné dans la Figure 2.3. L'instanciation partielle de départ est "53", indiquant qu'à ce stade de la résolution, deux variables sont déjà instanciées, l'une à 5, l'autre à 3. Le parcours de l'espace de recherche s'attache à affecter une valeur à la variable x_i, et lui attribue la valeur 4. Puis, la variable x_j est instanciée à la valeur 2. L'instantiation partielle qui en résulte, 5342, est alors détectée comme globalement inconsistante et son extension est interrompue. L'algorithme revient en arrière remet en cause la dernière extension de l'instantiation partielle, c.-à-d. $x_j \leftarrow 2$, et avance dans la recherche en évaluant l'affectation $x_j \leftarrow 3$. Ce processus d'extension/refutation de l'instanciation courante est répétée, jusqu'à trouver une solution ou en prouver l'absence. Les instanciations localement inconsistantes ne sont pas explorées plus en profondeur. Visuellement, un nœud blanc est un nœud dont toutes les extensions possibles non pas été encore totalement testées ; inversement, un nœud noir représente un nœud dont la preuve a été faite qu'il ne peut aboutir à aucune solution.*

En intégrant la propagation après chaque choix dans l'algorithme de backtrack, on obtient l'algorithme suivant (Algorithme 4). Lors de chaque appel à la méthode principale SEARCH, le réseau est localement consistant[4]. La première instruction (ligne 2) consiste à déterminer si au moins une variable demeure non instanciée parmi X. S'il existe au moins une variable non instanciée, il est alors nécessaire de créer une décision pour réduire l'espace de recherche (ligne 6). La construction de la décision (sélection de la variable et de la valeur à lui attribuer) n'est pas détaillée ici. Sa conception est souvent adaptée au problème traité.

Ensuite, une copie du réseau est effectuée (ligne 5). Cette copie permettra, si cela s'avère nécessaire, de restaurer le réseau dans l'état dans lequel il se trouvait avant la prise de décision δ et ainsi pouvoir propager sa réfutation au sein du réseau de contraintes. La réversibilité d'un réseau est gérée principalement de trois manières :

[4]On considère en effet que la consistance locale du réseau initial aura été préalablement vérifiée.

FIGURE 2.3 – Extrait d'arbre de recherche pour un CSP quelconque.



par copie : avant l'application d'une décision, une copie du réseau est faite ; cette approche *brute de force* atteints ses limites lorsque peu de modifications sont effectuées entre deux nœuds consécutifs de l'arbre de recherche.

par *trail* : ce système est un peu plus complexe : il se base sur le principe d'horodate ou *timestamp* d'une modification apportée au réseau ; ainsi seules les parties qui diffèrent d'un nœud à son successeur sont mémorisés. En cas de restauration, les opérations inverses des modifications mémorisées sont *dépilées* de la plus récente à la plus ancienne.

par recalcul : il s'agit d'une alternative à la copie, dans lequel la copie du réseau n'est faite que tous les n nœuds, entre temps, les décisions appliquées sont mémorisées. En cas de restauration d'un réseau, la copie la plus récente est restaurée et les décisions complémentaires sont appliquées pour recalculer le bon état.

De plus amples détails concernant la réversibilité d'un réseau sont donnés dans [1, 103]. On considère également que la récursivité de l'algorithme présenté crée une instance de cet algorithme avec ses propres variables locales. Puis, la décision δ est appliquée (ligne 6). L'application de cette décision retourne un ensemble de domaines $\mathcal{D}$ modifiés. Les événements induits remettent en cause la consistance locale des contraintes. En d'autres termes, il n'y a aucune garantie que toutes les contraintes soient à leurs points fixes. Pour retrouver un point fixe global, ou prouver que le réseau est globalement inconsistant, il est nécessaire de propager l'application de la décision au travers du réseau de contraintes. Cette opération est effectuée en exécutant la méthode PROPAGATE (ligne 7), décrite dans la section 2.2.1. Elle retourne la valeur logique `true` si un nouveau point fixe a été atteint, et la valeur logique `false` si une inconsistance a été détectée. Si un nouveau point fixe est atteint, la recherche se poursuit, en appelant récursivement la méthode SEARCH.

Autrement, il convient de réfuter la décision calculée au préalable (ligne 10). La copie des domaines $\mathcal{D}'$ est utilisée ici puisque $\mathcal{D}$ diffère de son état d'origine, au minimum par l'application de δ. L'événement de réfutation est propagé au sein du réseau de contraintes (ligne 11). Si une inconsistance locale est détectée, la procédure courante s'arrête et la procédure appelante reprend la main. Dans le cas contraire, il faut poursuivre la recherche (ligne 12). Dans le cas où toutes les variables sont instanciées, on se trouve en présence d'une solution, elle est enregistrée pour être restituée à la demande de l'utilisateur (ligne 15).

Algorithme 4 Algorithme de Recherche récursif.

```
1:  procedure SEARCH(X, D, C)
2:      Variable x ← CHOOSEVARIABLE(X, D)
3:      if x ≠ null then
4:          Decision δ ← COMPUTEDECISION(x, ⟨X, D, C⟩)
5:          (X, D', C') ← COPYOF(X, D, C)
6:          D ← APPLY(δ, D)
7:          if PROPAGATE(X, D, C) then
8:              SEARCH(X, D, C)
9:          end if
10:         D' ← NEGATE(δ, D')
11:         if PROPAGATE(X, D', C') then
12:             SEARCH(X, D', C')
13:         end if
14:     else
15:         RECORDSOLUTION(X, D)
16:     end if
17: end procedure
```

Si au moins une solution a été trouvée à la fin de l'exécution de la première instance de la procédure SEARCH, le CSP est satisfait. Si aucune solution n'a été trouvée, le réseau est alors insatisfait. Le pseudo code de l'Algorithme 4 calcule toutes les solutions. Une condition d'arrêt peut être ajoutée après l'enregistrement de la solution (ligne 15) pour ne pas avoir à parcourir toutes les solutions[5].

2.3.1 Exploration de l'arbre de recherche

La stratégie de branchement en soi à elle seule ne permet pas d'exprimer toutes les possibilités offertes par l'algorithme pour parcourir l'espace de recherche. En effet, tel qu'il a été présenté auparavant (Algorithme 4), l'algorithme de backtrack explore l'espace en profondeur d'abord, ou DFS (pour "*Depth First Search*" en anglais). Cette exploration est intéressante puisqu'elle offre un parcours dans lequel chaque nœud de l'arbre de recherche n'est visité qu'une seule fois. Des alternatives existent, dont certaines corrigent les imperfections de DFS. On peut mentionner le parcours en largeur d'abord, BFS ("*Breadth First Search*". Toutefois, bien qu'il soit largement utilisé en recherche opérationnelle pour résoudre des problèmes d'optimisation, ce parcours est peu utilisé en PPC parce qu'il sollicite beaucoup trop la mémoire. On peut également citer les explorations basées sur la notion de *divergence*, comme LDS [47, 58] ("*Limited Discrepancy Search*"), DDS [116] ("*Depth-bounded Discrepancy Search*") ou DBDFS [5] ("*Discrepancy-Bounded Depth First Search*"). Elles reposent toutes sur le principe qu'une stratégie de branchement adaptée fait peu d'erreurs. Donc, les chemins de décisions avec beaucoup de réfutations doivent être considérés comme moins intéressants. Un chemin de décisions allant du du nœud racine à une feuille qui ne comporte qu'une seule décision réfutée, *c.-à-d.* une seule divergence, devra être testé avant les chemins avec plus de divergences.

Si l'on souhaite résoudre un *problème d'optimisation sous contraintes*, il est opportun de s'inspirer des techniques de recherche locale. Un problème d'optimisation sous contraintes,

[5] De manière similaire, en ajoutant la création des coupes après l'enregistrement des solutions, cet algorithme peut être simplement adapté à la résolution de COP.

ou COP (pour "*Constraint Optimization Problem*") un est une extension du CSP : il est composé de variables, d'un domaine et de contraintes mais également d'une fonction objectif qui doit être optimisée. Il s'agit généralement d'un profit qu'il faut maximiser ou d'un coût qu'il faut minimiser. Le but est alors de chercher une solution *optimale*, *c.-à-d.* une solution qui optimise la fonction objectif sur l'ensemble de toutes les solutions. En PPC, il est commun de mémoriser le résultat de la fonction objectif dans une variable objectif.

En pratique, chaque nouvelle solution S impose une *coupe* sur la variable objectif, notée c_S, bornant son domaine. Une coupe impose que la prochaine solution doit être *meilleure* que la solution courante, jusqu'à ce que la solution optimale soit atteinte. Pour résoudre les COP, la technique de Recherche à Voisinage Large, qui s'inspire des techniques de recherche locale pour explorer les voisins d'une solution, a présenté de bons résultats.

La Recherche à Voisinage Large

Les techniques de Recherche Locale sont particulièrement efficaces pour résoudre les problèmes d'optimisation fortement combinatoires. La recherche locale consiste, à partir d'une solution, à parcourir tout ou partie des solutions *voisines* jusqu'à en trouver une qui soit améliorante voire optimale localement. La notion de voisinage doit être définie, et, majoritairement, elle est propre au problème traité. Par exemple, pour le problème du voyageur de commerce, le voisinage dit "2-opt" supprime deux arêtes dans la solution courante et reconnecte les deux tours ainsi formés. Bien entendu, il peut exister plusieurs définitions différentes de voisinage pour un même problème.

Naturellement, la programmation par contraintes s'est inspirée de ces techniques, sous la forme de Recherche à Voisinage Large, ou LNS (pour "*Large Neighborhood Search*", dont les bases ont été posées par Shaw en 1998 [86, 108]. La LNS est une métaheuristique conçue initialement pour calculer des mouvements adaptés au problème de tournées de véhicules. L'évaluation et la validation de ces mouvements étaient effectuées grâce à une recherche arborescente. La LNS est un algorithme en deux étapes qui *relaxe* partiellement une solution puis la *répare*. Étant donné une solution, une heuristique de voisinage est appelée, durant la phase de relaxation, pour construire un *voisin* à partir d'une solution. Un voisin est une instanciation partielle de la solution S courante : les domaines de certaines variables sont restaurés à leur état initial, alors que les variables restantes demeurent instanciées à leur valeur dans S. Cette notion de voisinage donne son nom à la technique. Il convient ensuite d'étendre l'instanciation partielle à une solution, et même s'il existe diverses manières de procéder[6], nous nous concentrons sur la technique d'origine dans laquelle la PPC est utilisée pour borner la variable objectif et instancier les variables non instanciées. S'il n'est pas possible d'étendre l'instanciation à une solution ou que la solution obtenue n'est pas améliorante, un nouveau voisin est construit, et la phase de réparation est lancée à nouveau. Si, au contraire, une nouvelle solution est trouvée, celle-ci sert alors de base pour une nouvelle relaxation. La LNS s'arrête lorsque l'optimalité d'une solution est prouvée[7] ou quand une limite de recherche est atteinte.

Bien que l'implémentation de la LNS soit simple, la difficulté principale réside dans la conception d'heuristiques de voisinage capables de faire avancer la recherche. En effet, il faut trouver un bon équilibre entre la diversification (*c.-à-d.* parcourir des sous-arbres inexplorés) et l'intensification (*c.-à-d.* les explorer entièrement). Le principe général de la LNS est décrit

[6] Il est, par exemple, possible d'utiliser la Programmation Mixte en Nombres Entiers, ou MIP (pour "*Mixed Integer Programming*"), comme cela a été fait dans [30].

[7] La LNS n'est pas conçue pour prouver l'optimalité d'une solution. On admettra, cependant, qu'il est possible de déléguer cette preuve d'optimalité, rendant la LNS complète.

dans l'Algorithme 5.

Algorithme 5 Large Neighborhood Search

Require: une solution S

1: **procedure** LNS
2: **while** Solution optimal non trouvée et critère d'arrêt non rencontré **do**
3: RELAX(S)
4: $S' \leftarrow$ FINDSOLUTION() ▷ La coupe c_S est postée
5: **if** $S' \neq$ NULL **then** ▷ Une solution améliorante a été trouvée
6: $S = S'$
7: **end if**
8: **end while**
9: **end procedure**

En partant d'une solution initiale S, la LNS sélectionne et relaxe un sous-ensemble de variables (RELAX, ligne 3). Une instanciation partielle est obtenue, et on va chercher à l'étendre vers une solution de meilleure qualité, du point de vue de la fonction objectif (ligne 4). Si aucune solution ne peut être obtenue sur la base de l'instanciation partielle (soit l'instanciation partielle ne peut être étendue, soit la solution obtenue n'est pas améliorante), une nouvelle instanciation partielle est générée. Si une solution S' améliorante est obtenue (ligne 5), elle est mémorisée (ligne 6). Ces opérations sont exécutées jusqu'à ce que la solution optimale soit construite ou que le critère d'arrêt (*p. ex.*, une limite de temps) soit atteint (ligne 2). La complétude de l'exploration de l'espace de recherche induite par l'Algorithme 5 repose sur le nombre des variables à relâcher décrit par la méthode RELAX, ainsi que sur la manière dont ce nombre évolue avec le temps. De plus, il est tout à fait envisageable d'autoriser des solutions de coûts équivalents (*walking* [84]), voire de dégrader la solution courante (comme cela peut-être fait dans une méthode de *recuit simulé* [57]) pour sortir des minima locaux .

Sélectionner les variables à relaxer est la partie délicate de l'algorithme. Un choix aléatoire des variables peut être considéré en premier lieu, mais il est rarement compétitif avec les heuristiques dédiées[43, 108]. Dans [108], Paul Shaw résout le problème de tournées de véhicules (Vehicle Routing Problem [27]) en sélectionnant l'ensemble des visites de clients à supprimer et à ré-insérer : en partant de tournées réalisables, certains clients sont sélectionnés pour ne plus être visités. Ils peuvent alors être ré-insérés ailleurs dans les tournées relaxées. Pour le problème de conception de réseau (Network Design Problem), la structure du problème est exploitée pour définir des voisinages pertinents [25]. Pour le problème d'ordonnancement d'ateliers à cheminements multiples (Job Shop Scheduling Problem), un voisinage qui intègre la fonction de coût a été étudié [30] ; les solutions partielles y sont réparées à l'aide d'un solveur MIP. Une autre approche repose sur un portefeuille de voisinages et des techniques d'apprentissage automatique pour sélectionner le voisinage le plus efficace. Cette approche a été évaluée avec succès sur des problèmes d'ordonnancement, avec un portefeuille de voisinages adaptés aux problèmes traités [61]. Dans l'article [84], Perron et Shaw testent plusieurs techniques pour améliorer le fonctionnement global de la LNS, sur la base d'un problème de production automobile en série (Car Sequencing Problem). La technique la plus intéressante, nommée *walking*, consiste à accepter des solutions intermédiaires équivalentes en terme de coût. Dans [70], les auteurs utilisent l'apprentissage par renforcement pour configurer dynamiquement la taille de la solution partielle, la limite de recherche et la sélection du voisinage. Ils ont comparé plusieurs configurations et ils ont pu conclure sur les deux premiers critères, mais le voisinage dédié décrit dans [84] reste toujours le plus compétitif pour le problème traité (une version modifiée du Car Sequencing Problem). Depuis

15 ans, la LNS a permis de s'attaquer à d'autres classes de problèmes : Service Technician Routing and Scheduling Problems [59], Pollution-Routing Problem [34], Founder Sequence Reconstruction Problem [98], Strategic Supply Chain Management Problem [29], Machine Reassignment Problem [73], pour n'en citer que quelques-uns.

Une autre approche consiste à définir des voisinages génériques. Dans [85], des voisinages sophistiqués, basés sur le graphe de dépendances entre variables, ont été proposés. Les auteurs présentent les voisinages guidés par la propagation (*propagation-guided*), dans lesquels le volume de réduction du domaine, au travers de la propagation, permet de lier les variables entre elles, à l'intérieur ou à l'extérieur des solutions partielles. À chaque appel de ce voisinage, une première variable est sélectionnée aléatoirement pour être faire partie de l'instanciation partielle, en lui affectant sa valeur dans la solution précédente. Cette instanciation partielle est propagée au travers du réseau de contraintes et un graphe de dépendance entre les variables est construit : les variables modifiées par propagation sont marquées. Chaque variable marquée non instanciée est placée dans une *liste à priorité*, où les variables sont triées en fonction de l'importance de la réduction que leur domaine a subi. La première variable de la liste est alors sélectionnée pour faire partie de l'instanciation partielle. La phase de sélection-propagation s'arrête lorsque la somme des logarithmes du domaine des variables est inférieure à une constante donnée. En pratique, ils proposent trois voisinages qui, combinés ensemble, permettent d'obtenir les meilleurs résultats pour le traitement du problème modifié de production automobile en série. Bien que générique, ces voisinages requièrent une fine paramétrisation : taille de la solution partielle, la manière dont elle évolue, qualification de la "dépendance" des variables entre elles. Cependant, cette approche combinée reste la référence quand il s'agit d'évoquer les voisinages génériques. Plus récemment, une autre approche générique a été publiée dans un workshop sur les techniques de recherche locale [71]. Les auteurs choisissent automatiquement parmi un portefeuille d'heuristiques, celle qui est appliquée pour fixer les variables, et obtiennent des résultats intéressants en comparaison avec une heuristique de choix aléatoire. Par contre, aucune comparaison n'a été faite avec les heuristiques guidées par la propagation. À notre connaissance, il n'existe d'autre publication basée sur cette idée.

La taille de la solution partielle est un paramètre important dans la conception des voisinages. Si le nombre de variables relaxées est trop important, la LNS repose exessivement sur la recherche arborescente : trouver une nouvelle solution dépend alors plus de la stratégie de branchement que du voisinage, et finalement souffre d'une faible diversification. Au contraire, si ce nombre est trop faible, la recherche arborescente aura tendance à parcourir un espace de recherche trop petit, et pourra rencontrer des difficultés à trouver des solutions. C'est pourquoi, Shaw [108] propose de renforcer la diversification en faisant évoluer la taille de la solution partielle, cela permet également de rendre cette approche complète.

Enfin, une contribution importante à l'efficacité d'une LNS a été décrite par Perron [83] : le redémarrage rapide, ou *fast restart* en anglais. Il a montré qu'imposer une petite limite de recherche (*p. ex.*, le nombre d'échecs rencontrés) lors de la phase de réparation était valable. Si la recherche ne trouve pas de solution avant d'atteindre la limite, une nouvelle solution partielle est générée et la recherche est relancée. Il vaut mieux diversifier la recherche en calculant rapidement un nouveau voisinage plutôt que de s'évertuer à réparer une solution, sans limite ni garantie de succès. Cette méthode limite le *trashing*, phénomène qui consiste à parcourir sans cesse des sous-arbres déjà explorés ne contenant aucune solution, et réduit la marge entre diversification et intensification. Les évaluations ont d'ailleurs montré que le *fast restart* permet d'améliorer la qualité des solutions trouvées.

Ces algorithmes s'attachent à parcourir, de manière appropriée, l'espace de recherche. Une alternative consiste à exploiter les échecs rencontrés pendant la recherche pour corriger

les mauvaises décisions et guider l'exploration.

2.3.2 Algorithmes de backtrack "intelligents"

Nogoods et explications sont utilisés depuis longtemps pour améliorer la recherche [42, 101, 89, 50, 114]. Une explication mémorise suffisamment d'information pour justifier d'une inférence faite lors de la propagation (réduction de domaine, échec, etc.). Nous introduisons la notion de *domaine initial*, noté $\mathcal{D}^i$, qui fait référence aux valeurs initialement disponibles pour chaque variable. On considérera alors un domaine $\mathcal{D}$ quelconque comme étant *plus petit* que le domaine initial, $\mathcal{D} \subseteq_X \mathcal{D}^i$. Étant donné une variable x_i et son domaine initial $D^i(x_i)$, la plus petite valeur de $D^i(x_i)$ est notée $\underline{D^i}(x_i)$, la plus grande valeur est notée $\overline{D^i}(x_i)$.

Définition 2.10 (Déduction) *Une* déduction *$(x \nleftarrow v)$ est la détermination qu'une valeur v doit être supprimée du domaine de la variable x.*

Définition 2.11 (Explication généralisée) *Une* explication généralisée*, g_expl$(x \nleftarrow v)$, de la déduction $(x \nleftarrow v)$ est définie par un ensemble de contraintes $C' \subseteq C$ et un ensemble de déductions Φ tels que $C' \wedge \Phi \wedge (x \leftarrow v)$ est globalement inconsistant.*

Une déduction s'explique généralement grâce à d'autres déductions. En remontant la chaîne d'inférences depuis une déduction, on retrouve certaines des décisions prises par la stratégie de recherche. Un Φ vide pour une déduction $x \nleftarrow v$ représente une déduction qui est soit directement due à l'application d'une décision sur x ou une déduction effectuée au nœud racine de l'arbre de recherche. Expliquer les déductions uniquement à l'aide de décisions est l'objectif des *explications*, il est possible de les calculer à partir d'explications généralisées.

Définition 2.12 (Explication) *Une* explication*, expl$(x \nleftarrow v)$, de la déduction $(x \nleftarrow v)$ est définie par un ensemble de contraintes $C' \subseteq C$ et un ensemble de décisions (instanciations) Δ tels que $C' \wedge \Delta \wedge (x \leftarrow v)$ est globalement inconsistant.*

D'une certaine façon, les explications fournissent une trace du comportement du solveur puisque chaque opération effectuée est expliquée [32].

Les explications font toujours référence au chemin de décisions. En expliquant le retrait de chaque valeur du domaine d'une variable, il est possible d'expliquer un échec (un domaine $D(x_i)$ vide) en calculant l'union des explications de chaque valeur retirée de $D^i(x_i)$. On obtient alors l'explication suivante : $expl(D(x_i) = \emptyset) = \bigcup_{j \in D^i(x_i)} expl(x_i \nleftarrow j)$. Le sous-ensemble de décisions obtenu est une instanciation qui ne peut être étendue à aucune solution, ce qui correspond à la définition d'un *nogood* [33]. Les nogoods sont les équivalents CSP des clauses apprises par les solveurs SAT [78]. La notion de nogood peut être généralisée en ne se limitant plus uniquement aux instanciations $(x \leftarrow v)$, mais également aux non-instanciations $(x \nleftarrow v)$. Les nogoods généralisés sont des explications relaxées dans lesquelles chaque chaîne d'inférence n'est pas nécessairement remontée jusqu'aux décisions.

Définition 2.13 (Nogood généralisé [56]) *Un* nogood généralisé *est un ensemble de contraintes $C' \subseteq C$, un ensemble de déductions (non-instanciations) Φ et un ensemble de décisions (instanciations) Δ tels que $C' \wedge \Phi \wedge \Delta$ est globalement inconsistant.*

Si une instanciation partielle recouvre un nogood, alors toutes les extensions de cette instanciation recouvriront ce nogood, et aucune ne pourra être étendue à une solution. Il convient

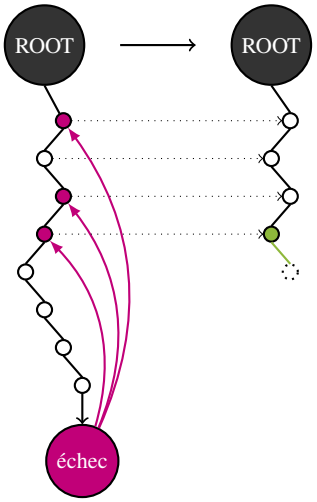
alors de ne pas explorer le sous-arbre induit et de revenir en arrière pour poursuivre la recherche. Les nogoods sont découverts au fur et à mesure de l'exploration de l'espace de recherche, lorsque des échecs sont rencontrés. Ils sont alors stockés dans une base de nogoods, qui agit comme une contrainte en filtrant des valeurs du domaine des variables présentes dans les nogoods. Les sous-espaces ne menant à aucune solution, lorsqu'ils sont détectés, sont donc mémorisés et évités en poursuivant l'exploration de l'espace de recherche.

En parallèle de l'alimentation de la base de nogoods, il est possible d'exploiter les explications afin d'effectuer des backtracks intelligents (non-chronologiques) pour améliorer la recherche. Des techniques complètes et incomplètes ont été proposées. Elles suivent toutes le même schéma : apprendre d'un échec en expliquant chaque modification de domaines et tirer profit de l'information réunie pour désigner les décisions à réfuter.

Avant de présenter ces techniques, nous rappelons qu'elles n'ont pas été étudiées comme des extensions de l'algorithme de backtrack décrit jusqu'ici. En effet, la nécessité de revenir en arrière réside dans le besoin de restaurer les domaines et les (états internes des) contraintes dans leur état avant l'application une décision quand celle-ci est réfutée. Cependant, c'est une faiblesse de cet algorithme. Puisque les explications retracent la chaîne logique de conséquences d'un événement, réfuter une décision revient à annuler les modifications qu'elle a entrainées, *c.-à-d.* remettre les valeurs supprimées dans les domaines. Ces opérations d'ajout ne sont jamais présentes dans les solveurs : elles impliquent de revoir la manière dont la réversibilité du réseau doit être gérée, notamment en autorisant l'ajout de valeurs dans le domaine des variables, en rendant les contraintes *décrementales*[8]. Toutefois, il est possible d'exploiter l'algorithme de backtrack pour éviter ces modifications complexes. C'est pourquoi nous décrivons les techniques d'amélioration de l'algorithme de backtrack comme s'exécutant en son sein.

Conflict-based Backjumping [33, 89] Cette première technique est complète. Lorsqu'un échec est rencontré, les décisions qui le justifie sont récupérées grâce aux explications. L'algorithme de recherche backtrack vers la décision la plus récente, elle est automatiquement réfutée, et la recherche se poursuit. L'explication de l'échec justifie la réfutation. Son schéma de fonctionnement est présenté dans la Figure 2.4. On parle de "saut en arrière" puisque la décision réfutée peut être éloignée de la décision dont l'application déclencha l'échec. Les décisions plus récentes que celles réfutées (donc n'expliquant pas l'échec) sont "oubliées", et une partie de l'apprentissage est alors perdue. La seconde méthode empêche ce comportement.

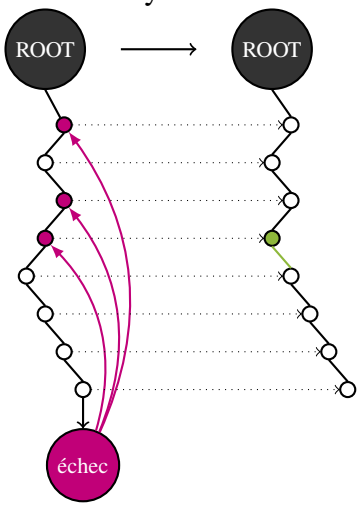
FIGURE 2.4 – Conflict-based Backjumping.



[8] Rendre décrementale une contrainte en extension revient à retrouver les supports précédemment supprimer pour les ré-autoriser lors de l'ajout d'une valeur dans le domaine d'une variable. Toutefois, pour les contraintes en intension, le fonctionnement n'est pas toujours aussi évident.

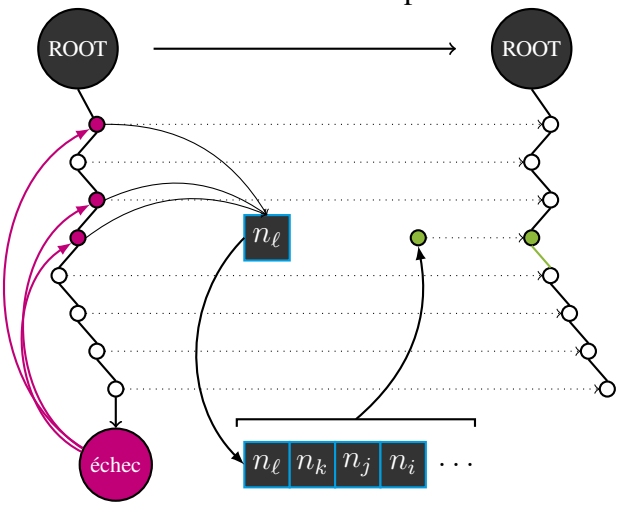
Dynamic Backtracking [42, 53] La seconde technique est également complète. Lorsqu'un échec est rencontré, les décisions qui l'expliquent sont récupérées grâce aux explications. Toutes les décisions du chemin de décisions sont maintenues, en respectant leur ordre d'application, excepté la décision la plus récente expliquant l'échec qui est réfutée. Et la recherche se poursuit. L'explication liée à l'échec justifie la réfutation. Cette technique comble la lacune de la technique précédente concernant l'apprentissage perdu. Son schéma de fonctionnement est présenté dans la Figure 2.5.

FIGURE 2.5 – Dynamic Backtracking.



Decision Repair [50, 87] La troisième technique, généralement incomplète [88], s'inspire des techniques de recherches locales. Lorsqu'un échec est rencontré, le nogood expliquant l'échec est calculé. Le nogood est ajouté à une liste tabou de nogoods. Cette liste de nogoods sert de base à la sélection de la décision à réfuter, et dont la négation est compatible avec les nogoods de la liste taboue. Son schéma de fonctionnement est présenté dans la Figure 2.6. La manière dont les décisions peuvent être sélectionnées et l'impact de ce choix sur l'efficacité de cette famille d'algorithmes ont été étudiés [88].

FIGURE 2.6 – Decision Repair.



Ces trois techniques suivent un schéma similaire : apprendre des échecs en associant à chaque modification de domaine une explication (fournie par le solveur) et exploitant cette information pour réduire l'espace de recherche. On évite alors le phénomène de trashing lorsque, par exemple, les premières décisions prises sont inconsistantes mais que la propagation n'est pas en mesure de le détecter précocement. La détection de l'échec est trivial à mettre en place, la difficulté majeure de ces techniques vient du calcul des explications (décrit dans [51]) et de la description des schémas d'explications pour les contraintes globales.

Exemple 2.8 (Explications) *Étant donné le problème défini par*

- *sept variables* x_1, x_2, x_3, x_4, x_5, x_6 *et* Ω,
- *leur domaine associé* $D(x_1) = [\![0,4]\!]$, $D(x_2) = [\![0,4]\!]$, $D(x_3) = [\![-1,3]\!]$, $D(x_4) = [\![-1,3]\!]$, $D(x_5) = [\![0,4]\!]$, $D(x_6) = [\![0,4]\!]$ *et* $D(\Omega) = [\![0,10]\!]$
- *et les contraintes suivantes* $C1 \equiv \sum_{i=1}^{6} x_i = \Omega$, $C2 \equiv x_1 \geq x_2$, $C3 \equiv x_3 \geq x_4$ *et* $C4 \equiv x_5 + x_6 > 3\rangle$

qui a pour objectif de minimiser la valeur attribuée à Ω*. Une première solution* $S_1 = \langle 0, 0, 2, 0, 2, 2, 6\rangle$ *est obtenue en appliquant le chemin de décisions* $P_{S_1} = (\delta_1, \delta_2, \delta_3, \delta_4, \delta_5)$*, où* $\delta_1 : \langle x_1, =, 0\rangle, \delta_2 : \langle x_4, =, 0\rangle, \delta_3 : \langle x_3, =, 2\rangle, \delta_4 : \langle x_5, =, 2\rangle$ *and* $\delta_5 : \langle x_6, =, 2\rangle$*.*

Le Tableau 2.2 donne la trace de la recherche. La première étape (Étape 1.a) décrit les ef-

TABLE 2.2 – Trace de la recherche d'une solution pour le COP définit dans l'Exemple 2.8.

Étape	Cause	Conséquences
1.a	δ_1	$x_1 = [\![0, \mathbf{0}]\!]$
1.b	$x_1 \wedge C2$	$x_2 = [\![0, \mathbf{0}]\!]$
2.a	δ_2	$x_4 = [\![0, \mathbf{0}]\!]$
2.b	$x_4 \wedge C3$	$x_3 = [\![\mathbf{0}, 3]\!]$
3.a	δ_3	$x_3 = [\![\mathbf{2}, \mathbf{2}]\!]$
3.b	$x_3 \wedge C1$	$\Omega = [\![\mathbf{2}, 10]\!]$
4.a	δ_4	$x_5 = [\![\mathbf{2}, \mathbf{2}]\!]$
4.b	$x_5 \wedge C4$	$x_6 = [\![\mathbf{2}, 4]\!]$
4.c	$x_5 \wedge x_6 \wedge C1$	$\Omega = [\![\mathbf{6}, \mathbf{8}]\!]$
5.a	δ_5	$x_6 = [\![\mathbf{2}, \mathbf{2}]\!]$
5.b	$x_6 \wedge C1$	$\Omega = [\![6, \mathbf{6}]\!]$

fets de l'application de δ_1 *:* x_1 *est instanciée à* 0*. Comme* x_1 *et* x_2 *sont liées par la contrainte* $C2$*, il est nécessaire de vérifier si* $C2$ *peut déduire de l'instanciation de* x_1 *des modifications du domaine de* x_2*. Par propagation de la contrainte* $C2$*,* x_2 *est instanciée à* 0 *également (Étape 1.b). Ensuite, l'application de* δ_2 *instancie* x_4 *à* 0 *(Étape 2.a), cela déclenche l'exécution du propagateur de* $C3$ *qui modifie la borne inférieure de* x_3 *à* 0 *(Étape 2.b). L'application de* δ_3 *instancie* x_3 *à* 2 *(Étape 3.a), la borne inférieure de* Ω *devient alors* 2*, à cause de l'application de* $C1$ *(Étape 3.b). L'application de* δ_4 *instancie* x_5 *à* 2 *(Étape 4.a), qui déclenche l'exécution du propagateur de* $C4$ *et change à* 2 *la borne inférieure de* x_6 *(Étape 4.b). Puis, le domaine de* Ω *est également mis à jour suite aux deux précédentes modifications (Étape 4.c). Enfin, l'application de* δ_5 *instancie* x_6 *à* 2 *(Étape 5.a). Cet événement instancie finalement* Ω *à la valeur* 6 *en exécutant le propagateur de* $C1$ *(Étape 5.b).*

Le Tableau 2.3 liste les explications calculées par valeurs retirées au cours de la résolution du COP, résultant en la solution S_1*. Une ligne pleine sépare les explications pour chaque variable. Une ligne en pointillé sépare les groupes d'explications, c.-à-d. les explications équivalentes pour plusieurs retraits de valeurs d'une variable. Dans cet exemple, certaines variables sont uniformément expliquées, p. ex., chaque retrait de valeur du domaine de* x_1 *est expliqué par* δ_1*. C'est également le cas pour les variables* x_2*,* x_4 *et* x_5*. Certaines explications ne sont pas aussi évidentes. Prenons la variable* Ω*, toutes les valeurs sont retirées par propagation (p. ex., la valeur* 0 *est expliquée par l'application de* δ_3 *et de la contrainte* $C1$*), certaines sont expliquées par plusieurs décisions et contraintes. Par exemple, l'explication du retrait de la valeur* 2 *du domaine de* Ω *n'est pas triviale. Rétrospectivement, l'accroissement de la borne inférieure de* Ω *s'explique de la manière suivante. La contrainte* $C1$ *a supprimé les valeurs* $[\![2, 5]\!]$ *de* Ω *parce que les bornes inférieures des variables* $x_1, \ldots, x_6$ *empêchent de lui affecter une valeur inférieure à 6. Parmi ces variables, seules* x_1 *et* x_2 *n'ont pas été modifiées depuis le début de la résolution et n'expliquent pas le domaine de* Ω*.*

- *La borne inférieure de* x_3 *a été modifiée deux fois : une première fois suite à la modification de la borne supérieure de* x_4 *et de l'application de* $C3$ *(Tableau 2.8, ligne 2.b).*

TABLE 2.3 – Explications par retrait de valeur pour toutes les variables du COP.

Variable	{Valeur retirée} ← Explication
x_1	$\{1\} \leftarrow \delta_1$,
	$\{2\} \leftarrow \delta_1$,
	$\{3\} \leftarrow \delta_1$,
	$\{4\} \leftarrow \delta_1$;
x_2	$\{1\} \leftarrow (\delta_1 \wedge C2)$,
	$\{2\} \leftarrow (\delta_1 \wedge C2)$,
	$\{3\} \leftarrow (\delta_1 \wedge C2)$,
	$\{4\} \leftarrow (\delta_1 \wedge C2)$;
x_3	$\{\text{-}1\} \leftarrow (\delta_2 \wedge C3)$,
	$\{0\} \leftarrow \delta_3$,
	$\{1\} \leftarrow \delta_3$,
	$\{3\} \leftarrow \delta_3$,
	$\{4\} \leftarrow \delta_3$;
x_4	$\{\text{-}1\} \leftarrow \delta_2$,
	$\{1\} \leftarrow \delta_2$,
	$\{2\} \leftarrow \delta_2$,
	$\{3\} \leftarrow \delta_2$,
	$\{4\} \leftarrow \delta_2$;
x_5	$\{0\} \leftarrow \delta_4$,
	$\{1\} \leftarrow \delta_4$,
	$\{3\} \leftarrow \delta_4$,
	$\{4\} \leftarrow \delta_4$;
x_6	$\{0\} \leftarrow (\delta_4 \wedge C4)$,
	$\{1\} \leftarrow (\delta_4 \wedge C4)$,
	$\{3\} \leftarrow \delta_5$,
	$\{4\} \leftarrow \delta_5$;
Ω	$\{0\} \leftarrow (\delta_3 \wedge C1)$,
	$\{1\} \leftarrow (\delta_3 \wedge C1)$,
	$\{2\} \leftarrow (\delta_2 \wedge \delta_3 \wedge \delta_4 \wedge C1 \wedge C3 \wedge C4)$,
	$\{3\} \leftarrow (\delta_2 \wedge \delta_3 \wedge \delta_4 \wedge C1 \wedge C3 \wedge C4)$,
	$\{4\} \leftarrow (\delta_2 \wedge \delta_3 \wedge \delta_4 \wedge C1 \wedge C3 \wedge C4)$,
	$\{5\} \leftarrow (\delta_2 \wedge \delta_3 \wedge \delta_4 \wedge C1 \wedge C3 \wedge C4)$,
	$\{7\} \leftarrow (\delta_1 \wedge \delta_2 \wedge \delta_3 \wedge \delta_4 \wedge \delta_5 \wedge C1 \wedge C2)$,
	$\{8\} \leftarrow (\delta_1 \wedge \delta_2 \wedge \delta_3 \wedge \delta_4 \wedge \delta_5 \wedge C1 \wedge C2)$,
	$\{9\} \leftarrow (\delta_1 \wedge \delta_2 \wedge \delta_3 \wedge \delta_4 \wedge C1 \wedge C2)$,
	$\{10\} \leftarrow (\delta_1 \wedge \delta_2 \wedge \delta_3 \wedge \delta_4 \wedge C1 \wedge C2)$;

La modification de la borne supérieure de x_4 s'expliquant par δ_2 (Tableau 2.8, ligne 2.a). La deuxième fois, à cause de l'application de la décision δ_3 (Tableau 2.8, ligne 3.a).

- *La borne inférieure de x_5 s'explique par l'application de la décision δ_4.*
- *Enfin, la borne inférieure de x_6 s'explique par l'exécution de la contrainte $C4$ sur modification du domaine de x_5 causé par δ_4.*

Donc, le retrait de la valeur 2 du domaine de Ω s'explique par l'application de $\delta_2 \wedge \delta_3 \wedge \delta_4 \wedge C1 \wedge C3 \wedge C4$.

Composants clés d'un système d'explication Instrumenter un solveur de contraintes avec des explications requiert de considérer les fonctionnalités suivantes :

- Calculer les explications : les réductions de domaine sont habituellement associées à une *cause* : le propagateur qui a effectivement fait la modification. Cette information est utilisée pour calculer l'explication. Ceci peut être fait de manière synchrone, pendant la propagation (une modification intrusive de l'algorithme de propagation), ou asynchrone, après la propagation (en appelant un service fourni par le propagateur).
- Stocker les explications : une structure de données est nécessaire pour être capable de stocker les décisions prises par le solveur, les réductions de domaines et leurs explications. Il existe plusieurs manières de gérer le stockage des explications : un stockage *à plat* des modifications de domaines et de leurs explications composé de propagateurs et des décisions prise, ou un stockage *arborescent* des modifications de domaines et de leurs explications exprimées au travers des précédentes modifications de domaines [32].
- Accéder aux explications : la structure de données utilisée pour stocker les explications doit donner un accès, non seulement aux explications relatives aux modifications de domaines, mais également aux bornes courantes des domaines, aux valeurs retirées, etc.

Bien que leur utilité ne soit plus à démontrer, l'intégration d'un système d'explications dans un solveur de contraintes souffre de plusieurs maux :

- Consommation mémoire : stocker les explications requiert de stocker, d'une manière ou d'une autre, toutes les modifications que subissent les variables ;
- Consommation CPU : calculer les explications se fait habituellement avec un coût, même si l'algorithme de propagation peut être exploité partiellement ;
- Génie logiciel : l'instrumentation d'un solveur avec les explications se fait habituellement de manière intrusive.

Depuis une dizaine d'années, une alternative au système d'explications tel que nous l'avons décrit émerge. Il s'agit d'une approche dans laquelle un solveur de contraintes « traditionnel » embarque un solveur SAT, cette hybridation forme un *solveur à génération paresseuse de clauses* [79, 111, 28] ("*Lazy clause generation solver* en anglais). Un solveur SAT [78] (pour "*boolean Satisfiability problem* en anglais) est un solveur spécifique qui encode uniquement des variables booléennes. Il dispose d'un algorithme de recherche exploitant les conflits, et ne traite les contraintes que sous la forme de clauses (proposition

disjonctive de littéraux), *c.-à-d.* $l_1 \vee l_2 \vee \ldots \vee l_n$ où l_i est un *littéral* pouvant prendre les valeurs `true` ou `false`. Dans l'approche hybride, le solveur de contraintes sert de base à la modélisation, à l'aide de variables entières, et la recherche arborescente. Cependant, les propagateurs du solveur de contraintes ne filtrent plus directement les domaines des variables, mais génèrent des clauses pour le solveur SAT. Pour cela, un lien entre les variables entières et les variables booléennes du solveur SAT est créé. En conséquence, le domaine d'une variable x, $D(x) = \{v_1, \ldots, v_k\}$, est représenté à l'aide de $2k$ variables booléennes $[\![x = v_1]\!], \ldots [\![x = v_k]\!]$ et $[\![x \leq v_1]\!], \ldots, [\![x \leq v_k]\!]$. La variable booléenne $[\![x = v]\!]$ prend pour valeur `true` si la variable entière x est instanciée à la valeur v, et prend pour valeur `false` si x prend une valeur différente de v. Un raisonnement similaire s'applique pour $[\![x \leq v]\!]$ et simule le raisonnement aux bornes. Les clauses générées par les propagateurs sont ajoutées au solveur SAT. Ce dernier déduit de ces clauses des instanciation des variables booléennes, qui induisent des modifications des variables entières. Les échecs sont également expliqués et ces explications sont ajoutées au solveur SAT pour élaborer des nogoods [74]. Le solveur de contraintes est alors en mesure de faire un « saut en arrière » [78], ou *backjumping*. La résolution profite alors automatiquement de ses différentes techniques et obtient de bons résultats sur une grande variétés de problèmes.

2.4 Synthèse

Dans ce chapitre, nous avons cherché à donner une image globale de la programmation par contraintes. Nous allons maintenant nous attarder plus particulièrement sur trois d'entre eux : la recherche à voisinage large, le système d'explications et la propagation des contraintes. Nos études s'inscrivent dans une démarche d'enrichissement de la bibliothèque de techniques de la programmation par contraintes pour répondre aux besoins des utilisateurs quelle que soit leur expérience.

Dans une première partie, nous allons exploiter les explications pour offrir des heuristiques de voisinages génériques et efficaces pour la recherche à large voisinage. Nous pensons que les explications constituent une source d'information sur la structure du problème qu'il est possible d'exploiter autrement que pour agir sur l'algorithme de backtrack. Pour cela, nous tâcherons d'alléger le processus de calcul des explications pour qu'il soit utilisable en pratique. Nous proposerons deux heuristiques de conception de voisinages basés sur les explications, et montrerons qu'ils sont compétitifs avec les heuristiques génériques références sur un ensemble varié de problèmes extrait de la librairie MiniZinc. Nous compléterons ainsi l'arsenal de méthodes typées « boîte noire » déjà disponibles dans les solveurs modernes pour résoudre les problèmes.

Dans une seconde partie, nous étudierons le mécanisme de propagation et proposerons une solution, sur la base d'un langage dédié, qui donne accès sa personnalisation dès la phase de modélisation. Nous souhaitons ouvrir cette partie du solveur à la modélisation pour donner la possibilité aux développeurs et modeleurs avancés de prototyper des schémas de propagation. Ce processus d'élaboration d'algorithme de propagation doit se faire sans que les utilisateurs aient à se préoccuper de sa mise en œuvre, des structures de données utilisées aux propriétés et garanties qu'il offre habituellement. L'objectif est d'enrichir la déclarativité propre à la programmation par contraintes.

II

Une Recherche à Voisinage Large à base d'Explications

3

Introduction

> « *We strongly believe that explanations for constraint programming is a really hot topic for the future.* »
>
> (Narendra Jussien, 2003, *The versatility of using explanations within constraint programming* [51])

La programmation par contraintes s'est toujours inspirée de domaines qui lui sont connexes, tels que l'Intelligence Artificielle ou le Calcul Formel, et les techniques de Recherche Locale ne font pas exception. Ces techniques sont particulièrement efficaces quand il s'agit de résoudre des problèmes d'optimisation fortement combinatoires, et c'est pourquoi elles ont été adaptées à la programmation par contraintes. La technique de recherche locale sans doute la plus connue et la plus utilisée en programmation par contraintes est celle de Recherche à Voisinage Large [108, 86], ou LNS (pour "*Large Neighborhood Search*" en anglais). La LNS est une technique dont l'intégration dans un solveur de contraintes est relativement simple : elle exploite la programmation par contraintes pour évaluer et valider des voisins générés à l'aide d'heuristiques. La LNS a montré son efficacité sur un grand nombre de problèmes variés, du problème de conception de réseaux au problème d'ordonnancement d'ateliers en passant par le problème de tournées de véhicules, à l'aide d'heuristiques de voisinages dédiés aux problèmes traités. C'est d'ailleurs la principale limite d'une telle approche, qui vient en contradiction avec l'esprit de la programmation par contraintes, "*the user states the problem, the computer solves it.*" (Eugène C. Freuder). De rares propositions d'heuristiques de voisinages génériques ont été proposées ([85, 71]). Malheureusement, elles n'ont été évaluées que sur le problème de Car Sequencing, et seule celle proposée par Perron et autres [85] a été montrée comme étant substituable aux heuristiques de voisinage spécifiques. Ce voisinage fonctionne de la manière suivante : une première variable est sélectionné pour faire partie du voisinage, elle est instanciée à sa valeur dans la solution précédente et cette modification est propagé dans le réseau de contraintes. Les variables modifiées par effet de bord sont alors retenues pour être choisie à leur tour. La constitution du voisinage s'arrête lorsque la somme des logarithmes du domaine des variables est inférieure à une constante. Les autres proposent deux autres alternatives à ce voisinage, également basés sur la propagation. La conception de ce voisinage est intéressante à plus d'un titre : d'abord parce

que le voisinage est générique, et ne nécessite “que” d’être paramétrée. Mais surtout parce qu’il détourne, pour l’utiliser autrement, une fonctionnalité centrale de l’outil sous-jacent puisque l’algorithme de propagation embarqué dans la PPC sert à construire un graphe de dépendance entre les variables d’un problème. L’adaptation au problème traité ne se fait plus explicitement, à l’aide d’une heuristique de voisinage dédié, étant donné que cet algorithme “découvre”, au travers de la propagation, la structure du problème et conçoit des voisins adaptatifs.

Une autre manière de découvrir la structure interne d’un problème et d’en tirer parti pour mieux résoudre un problème est d’utiliser les explications [23]. Les explications retracent le mécanisme de propagation, et identifient les contraintes et décisions responsables de l’état courant du domaine d’une variable. Elles sont ensuite communément exploitées sur des conflits, pour expliquer l’absence de solution dans une branche de l’arbre de recherche. Elles doivent permettre de réorienter l’exploration de l’espace de recherche, en apprenant sur les erreurs. Ces dernières années, les techniques basées sur les explications ont connu un regain d’intérêt en programmation par contraintes, notamment avec l’implémentation de solveurs générant paresseusement des clauses [79], hybridant un solveur de contraintes avec un solveur SAT. Malgré cela, et bien que de plus en plus de solveurs de contraintes soient disponibles au téléchargement, aucun n’intègre de système d’explications avancé. Cette situation trouve ses origines dans les deux composantes suivantes. En premier lieu, il existe peu d’études, proportionnellement aux autres domaines constitutifs de la programmation par contraintes, sur l’exploitation des explications dans les solveurs modernes. En grande partie, les explications (et les nogoods) ont été évaluées en interaction avec des algorithmes de *Forward Checking* [45], principe aujourd’hui largement remplacé par celui MAC – *Maintaining Arc Consistency*. Il en découle une méconnaissance des explications. En second lieu, il faut noter, non pas les difficultés d’implémentation des explications dans un solveur de contraintes –d’autres éléments d’un solveur sont également complexes à développer correctement–, mais plutôt son aspect intrusif. Chacune des modifications des variables doit être tracée, en instrumentant les variables, et justifiée, en équipant les contraintes de schémas d’explications. Et bien que ces schémas puissent être généralisables, rien ne vaut une implémentation spécifique. Ces deux constats, fortement corrélés, freinent la diffusion des explications dans les solveurs de contraintes modernes et leurs vulgarisations. Face à ce bilan, il est légitime, pour le développeur d’un solveur, de s’interroger sur la rentabilité d’implémenter un système d’explications dans son outil. De plus, les explications, tout comme la programmation par contraintes elle-même, est plus encline au traitement de problèmes de satisfaction plutôt que d’optimisation. L’ajout de coupe au cours de la résolution peut invalider des explications et affaiblir la découverte de la structure du problème.

Nous sommes persuadés que le mariage des explications et de la recherche à voisinage large est, sans doute, une réponse aux défauts des deux approches. Les explications doivent permettre d’exploiter la structure interne du problème traité, autorisant la conception de voisinages génériques et ne requérant pas de paramétrage pour la LNS. La LNS doit permettre de limiter la consultation des explications aux seules solutions, réduisant ainsi les coûts liés à leurs calculs. L’objectif d’une telle collaboration est, bien entendu, de rendre la résolution des problèmes d’optimisation plus efficace.

Dans cette partie, nous montrons que des voisinages génériques et efficaces, difficiles à trouver, pour LNS peuvent être construits à l’aide des explications. Une première contribution est de proposer deux voisinages sans configuration nécessaire pour construire des voisins d’une solution, l’un est basé sur l’explication de l’incapacité à réparer une solution partiellement détruite, l’autre sur l’explication de la nature non-optimale de la solution courante. Une deuxième contribution consiste en la mise en œuvre et l’évaluation de ces voisinages. Nous

indiquons également comment un système d'explications léger et utilisable en pratique peut être mis en place, comme il l'a été dans le solveur de contraintes Choco [90, 91]. Nous évaluons nos propositions sur un ensemble de problèmes d'optimisation extraits du Challenge MiniZinc [36]. Nous comparons nos approches avec une autre approche générique, guidée par la propagation, et montrons qu'elles permettent d'obtenir d'aussi bonnes solutions, voire de meilleures, posant ainsi les bases d'une nouvelle manière d'utiliser les explications pour améliorer la recherche.

Le Chapitre 4 est consacré à la description des voisinages. La section 4.1 décrit un premier voisinage, basé sur les explications du conflit existant entre le chemin de décisions menant à une solution et la coupe qui en découle. Nous présentons l'algorithme permettant de l'implémenter et déroulons un exemple d'application. La section 4.2 présente le second voisinage, qui repose sur les explications liées à la nature non-optimale de la solution courante en déterminant quelles sont les décisions qui ont empêché d'obtenir une solution de meilleure qualité. L'algorithme permettant de l'implémenter et un exemple d'application y sont également décrits. Des informations complémentaires aux descriptions des voisinages, comme la manière dont les explications peuvent être intégrées dans un solveur de contraintes, sont présentés dans la section 4.3. Le Chapitre 5 est dédié à l'évaluation des voisinages. Après avoir décrit les différents voisinages évalués, le protocole expérimental et les instances traitées (Section 5.1), une première évaluation comparative avec le voisinage proposé par Perron et autres [85] est faite (Section 5.2). Puis, nous proposons plusieurs combinaisons de ces voisinages entre eux et avec les voisinages de Perron et autres (Section 5.2.4 et Section 5.3). Enfin, une analyse plus détaillée, sur quelques instances choisies, est présentée dans la section 5.4.

4

Voisinages basés sur les explications

Dans cette Section, nous introduisons deux nouvelles techniques de calculs de voisinages basées sur les explications pour le framework LNS. Ces voisinages sont appelés `exp-cft` et `exp-obj`. Ce sont des implémentations particulières de la méthode RELAX(S) présentée dans l'Algorithme 5 (Section 2.3.1). Pour ne pas les alourdir inutilement, les descriptions suivantes seront positionnées dans un contexte de minimisation. Mais, de simples modifications permettent de les adapter à un contexte de maximisation.

Préliminaires

En pratique, le domaine d'une variable est rarement défini à l'aide d'une contrainte unaire. Il est usuel, soit de lister les valeurs de départ, on dit alors que le domaine est *énuméré*, soit de symboliser le domaine à l'aide de deux entiers, l'un représentant la borne inférieure initiale, l'autre, la borne supérieure initiale, on dit alors que le domaine est *borné*. En plus d'impacter le filtrage des contraintes, le choix de la représentation des domaines a une incidence sur l'espace mémoire occupé par un modèle. D'ordinaire, le domaine initial de la variable objectif est défini sur $\mathbb{Z}$, sans véritable restriction, on supposera donc qu'il est borné.

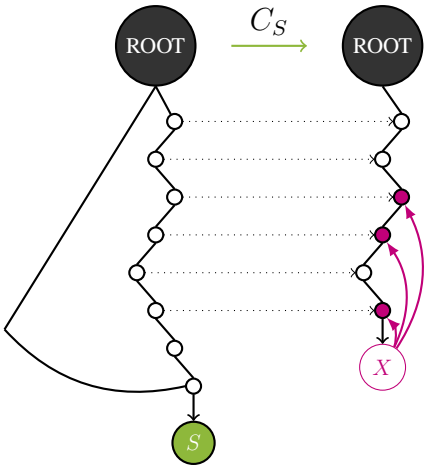
Hypothèse 4.1 *Le domaine de la variable objectif est borné.*

Les conséquences de la réfutation de cette hypothèse sont discutées dans la section 4.3.4.

4.1 Expliquer la coupe : `exp-cft`

Historiquement, les explications ont été utilisées pour expliquer un conflit dans l'optique de le réparer. En conséquence, le premier voisinage que nous allons décrire repose sur l'explication des conflits. Cependant, plutôt que d'expliquer les conflits rencontrés au cours de la phase de réparation d'un voisin, nous allons forcer la génération d'un conflit sur la base d'une solution pour générer des voisins. Nous partons du constat qu'il y a largement moins de solutions que d'échecs dans la résolution d'un problème, et donc, que le surcoût induit par l'utilisation des explications en sera réduit d'autant. Lors de la résolution d'un problème

FIGURE 4.1 – Schéma du voisinage `exp-cft`.



d'optimisation, chaque nouvelle solution impose une coupe établissant que la valeur affectée à la variable objectif doit être strictement inférieure dans la prochaine solution qu'elle n'est dans la solution courante, et ce, jusqu'à ce que la valeur optimale soit atteinte. Ainsi, pour une solution S donnée, une fois la coupe c_S imposée, il n'est plus possible d'appliquer toutes les décisions du chemin B_S menant à S sans générer un conflit. En effet, les appliquer entièrement va contraindre la variable objective a être instanciée à $I[\Omega]$ alors que cette valeur a été supprimée par la coupe c_S. Connaitre les décisions qui sont en conflit avec la dernière coupe imposée vont permettrent certainement de créer un voisin facilement réparable, faisant alors progresser la recherche vers la solution optimale. Ce raisonnement sert de base au premier voisinage, nommé `exp-cft`. Il est schématiquement présenté dans la Figure 4.1.

Tout d'abord, un conflit est forcé en appliquant les décisions de B_S en même temps que la coupe c_S est imposée. Lorsque le conflit est généré, les décisions en relation avec ce conflit sont extraites de la base d'explications. Puis, les voisins sont calculés sur la base des variables associées aux décisions en conflit avec la coupe. Puisque l'application de la totalité des décisions de B_S menait à une solution, *c.-à-d.* une affectation valide, relaxer certaines décisions du chemin de décisions laissera certaines variables non instanciées. La méthode complète est décrite dans l'Algorithme 6, le point d'entrée se fait par la méthode `RELAX_EXP-CFT`.

À chaque fois qu'une nouvelle solution est trouvée (ligne 3), la méthode EXPLAINCUT (ligne 4) est appelée. Elle retourne l'explication du conflit, et les décisions expliquant le conflit sont extraites (ligne 5). Puis, certaines de ces décisions sont sélectionnées aléatoirement (ligne 7) pour être supprimées du chemin de décision B_S. Enfin, le chemin relaxé est appliqué (ligne 8).

La méthode EXPLAINCUT (lignes 10-19) fonctionne de la manière suivante : tout d'abord, le chemin de décision B_S menant à la solution S est extrait (ligne 11), et la coupe est imposée (ligne 12). Ensuite, toutes les décisions de B_S sont appliquées et propagées une à une (lignes 14-17), en respectant l'ordre original, *c.-à-d.* chronologique. Lorsque le problème devient insatisfait (ligne 17), la boucle est stoppée, et les décisions en relation avec le conflit sont extraites de la base d'explications (ligne 18) à l'aide de la méthode EXPLAINCONFLICT.

La méthode EXPLAINCONFLICT (lignes 20-30) consulte la base d'explications et renvoie l'ensemble des décisions et contraintes qui expliquent le conflit. Le type de conflit rencontré conditionne la manière dont les explications sont calculées. Si le domaine d'une variable est vidé (ligne 22), l'explication qui en résulte est la conjonction des explications de chacun des retraits de valeur de la variable vidée (via un appel à la méthode EXPLAINREMOVAL,

Algorithme 6 Voisinage guidé par la coupe (dans un contexte de minimisation)

Require: Ω : la variable objectif
Require: k : un entier
Require: Δ_c : ensemble des décisions liées au conflit avec la coupe
1: **procedure** RELAX_EXP-CFT(S)
2: $B \leftarrow$PATHTO(S) ▷ Extrait le chemin de décisions menant à S
3: **if** une nouvelle solution a été trouvée **then**
4: $\mathcal{E} \leftarrow$ EXPLAINCUT(S, Ω)
5: $\Delta_c \leftarrow$EXTRACTDECISION($\mathcal{E}$) ▷ Extrait les décisions des explications
6: **end if**
7: $R \leftarrow$RANDOM(Δ_c) ▷ Sélectionne aléatoirement des décisions de Δ_c
8: APPLY($B \setminus R$) ▷ Applique B moins R
9: **end procedure**
10: **procedure** EXPLAINCUT(S, Ω)
11: $B \leftarrow$ PATHTO(S)
12: $D^i(\Omega) \setminus I[\Omega]$ ▷ Impose la coupe
13: $\mathcal{F} \leftarrow \emptyset$ ▷ État de la propagation
14: **repeat**
15: $\delta \leftarrow$POLLFIRST(B)
16: $\mathcal{F} \leftarrow$ APPLY(δ) ▷ δ appliquée et propagée
17: **until** $\mathcal{F} \neq \emptyset$ ▷ Un conflit est détecté par propagation
18: **return** EXPLAINCONFLICT($\mathcal{F}$)
19: **end procedure**
20: **procedure** EXPLAINCONFLICT($\mathcal{F}$)
21: $\mathcal{E} \leftarrow \emptyset$ ▷ Explication du conflit
22: **if** $\mathcal{F} \equiv$ le domaine de x est vide **then**
23: **for** $v \in D(x)$ **do**
24: $\mathcal{E} \leftarrow \mathcal{E} \bigcup$EXPLAINREMOVAL($D^i(x), v$) ▷ Explique le retrait de v de $D(x)$.
25: **end for**
26: **else if** $\mathcal{F} \equiv c$ est inconsistante **then**
27: $\mathcal{E} \leftarrow$EXPLAINCONSTRAINT(c) ▷ Peut être contrainte-specifique
28: **end if**
29: **return** $\mathcal{E}$
30: **end procedure**

ligne 24). Si une contrainte est inconsistante (ligne 26), la méthode EXPLAINCONSTRAINT est appelée pour constituer l'explication (ligne 27). Par défaut, cette méthode appelle la méthode EXPLAINREMOVAL pour chacune des valeurs retirées du domaine des variables impliquées dans la contrainte. Il est possible de proposer une explication plus précise en adaptant le calcul de l'explication à la contrainte, en spécifiant la méthode EXPLAINCONSTRAINT. Finalement, les méthodes EXPLAINREMOVAL et EXPLAINCONSTRAINT consultent la base d'explications.

Étant donné Δ_c l'ensemble des décisions en relation avec le conflit, il y a $2^{|\Delta_c|-1}$ sous-ensembles de l'ensemble Δ_c, chacun d'entre eux correspondant à une relaxation possible de B_S. Énumérer tous les sous-ensembles de Δ_c n'est pas polynomial, et il n'est pas garanti que les plus petites instanciations partielles puissent être étendues plus facilement vers une solution que les grandes. Donc, pour favoriser la diversification de `exp-cft`, et pour s'autoriser à tester des voisinages de tailles diverses, nous choisissons aléatoirement α décisions à relaxer, où α est choisi aléatoirement dans $[1, |\Delta_c| - 1]$ (Algorithme 6, Ligne 7).

Exemple 4.1 (`exp-cft`) *Étant donné le problème défini auparavant (Exemple 2.8), dont nous rappelons les caractéristiques :*

- *sept variables* x_1, x_2, x_3, x_4, x_5, x_6 *et* Ω,
- *leur domaine associé* $D(x_1) = [\![0, 4]\!]$, $D(x_2) = [\![0, 4]\!]$, $D(x_3) = [\![-1, 3]\!]$, $D(x_4) = [\![-1, 3]\!]$, $D(x_5) = [\![0, 4]\!]$, $D(x_6) = [\![0, 4]\!]$ *et* $D(\Omega) = [\![0, 10]\!]$
- *et les contraintes suivantes* $C1 \equiv \sum_{i=1}^{6} x_i = \Omega$, $C2 \equiv x_1 \geq x_2$, $C3 \equiv x_3 \geq x_4$ *et* $C4 \equiv x_5 + x_6 > 3\rangle$

qui a pour objectif de minimiser la valeur attribuée à Ω*. Une première solution* $S_1 = \langle 0, 0, 2, 0, 2, 2, 6\rangle$ *est obtenue en appliquant le chemin de décisions* $P_{S_1} = (\delta_1, \delta_2, \delta_3, \delta_4, \delta_5)$*, où* $\delta_1 : \langle x_1, =, 0\rangle, \delta_2 : \langle x_4, =, 0\rangle, \delta_3 : \langle x_3, =, 2\rangle, \delta_4 : \langle x_5, =, 2\rangle$ *and* $\delta_5 : \langle x_6, =, 2\rangle$*.*

Sur un appel de la méthode EXPLAINCUT *du voisinage* `exp-cft` *(Algorithme 6, Ligne 4), la première instruction est d'imposer la coupe (ici,* $c_{S_1} \equiv \Omega < 6$*), puis les décisions de* B_{S_1} *sont appliquées. La trace d'exécution est reportée dans la Table 4.1.*

TABLE 4.1 – Trace de la recherche du problème d'optimisation, défini dans l'exemple 2.8, avec le voisinage `exp-cft`.

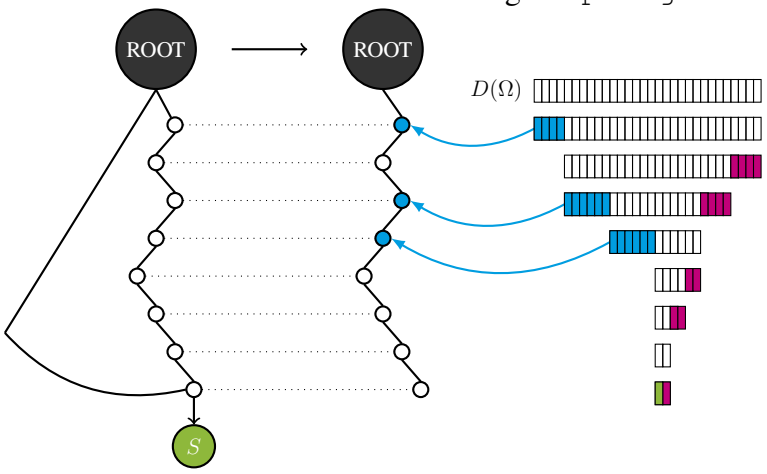
Étape	Cause	Consequences
1'.a	C_{S_1}	$\Omega = [\![0, \mathbf{5}]\!]$
2'.a	δ_1	$x_1 = [\![0, \mathbf{0}]\!]$
2'.b	$x_1 \wedge C2$	$x_2 = [\![0, \mathbf{0}]\!]$
3'.a	δ_2	$x_4 = [\![0, \mathbf{0}]\!]$
3'.b	$x_4 \wedge C3$	$x_3 = [\![\mathbf{0}, 3]\!]$
4'.a	δ_3	$x_3 = [\![\mathbf{2}, \mathbf{2}]\!]$
4'.b	$x_3 \wedge C1$	$x_5 = [\![0, \mathbf{3}]\!], x_6 = [\![0, \mathbf{3}]\!], \Omega = [\![\mathbf{2}, 5]\!]$
4'.c	$x_5 \wedge x_6 \wedge C4$	$x_5 = [\![\mathbf{1}, 3]\!], x_6 = [\![\mathbf{1}, 3]\!]$
4'.d	$x_5 \wedge x_6 \wedge C1$	$x_5 = [\![1, \mathbf{2}]\!], x_6 = [\![1, \mathbf{2}]\!], \Omega = [\![\mathbf{4}, 5]\!]$
4'.e	$x_5 \wedge x_6 \wedge C4$	$x_5 = [\![\mathbf{2}, 2]\!], x_6 = [\![\mathbf{2}, 2]\!]$
4'.f	$x_5 \wedge x_6 \wedge C1$	inconsistance

L'application de la coupe supprime les valeurs supérieures à 5 *du domaine de la variable objectif* Ω *(Ligne 1'.a). L'application de* δ_1 *(Ligne 2'.a et Ligne 2'.b) et* δ_2 *(Ligne 3'.a et Ligne 3'.b) ont le même effet avec ou sans la coupe. Cependant, l'application de* δ_3 *(Ligne 4'.a), avec la coupe* c_{S_1}*, déclenche plus de modifications et se termine en échec causé par l'exécution du propagateur de* $C1$ *(Ligne 4'.f). Les domaines des variables* x_i *et* Ω *sont tels qu'il est impossible de trouver une valeur d'instanciation pour* Ω *à cause de la contrainte* $C1$*. En effet, dans leur article sur la contrainte* `Scalar` *[46], Harvey et Schimpf ont établi que si* $\sum_{i=1}^{n} \overline{D}(x_i) - \underline{D}(\Omega) > 0$ *alors il n'est pas possible de satisfaire la contrainte . Dans notre cas,* $\sum_{i=1}^{n} \overline{D}(x_i) - \underline{D}(\Omega) = (0+0+2+0+2+2) - 4 = 2 > 0$*, donc, l'inconsistance s'explique par les bornes supérieures courantes des* x_i *et la borne inférieure courante de* Ω*. L'état du domaine de ces variables s'expliquent par les décisions* δ_1*,* δ_2 *et* δ_3*. Les décisions en relation avec le conflit sont :* $\Delta_c = \{\delta_1, \delta_2, \delta_3\}$*. Les décisions* δ_4 *et* δ_5 *n'ont pas été appliquées avant que le conflit ne soit détecté. Donc, elles ne peuvent pas faire partie des décisions expliquant le conflit.*

Lors d'un appel à la méthode RELAX_EXP-CFT*, les décisions* δ_4 *and* δ_5 *seront maintenues par défaut, avec également deux décisions, ou plus, choisie aléatoirement parmi* Δ_c*.*

L'explication d'un conflit peut ne pas être unique ni minimale [51]. Il peut n'y avoir aucun voisin calculé grâce à `exp-cft` qui puisse être étendu à une solution. Cependant, si l'application de la coupe échoue directement, alors Δ_c sera forcément vide et la résolution peut être interrompue : la solution optimale a été trouvée et prouvée (Algorithme 6, Ligne 2).

FIGURE 4.2 – Schéma du voisinage exp-obj.



4.2 Expliquer le domaine de la variable objectif : exp-obj

La méthode précédente définit des voisinages basés sur les conflits, ce qui est la manière habituelle d'exploiter les explications. Nous présentons maintenant un voisinage alternatif basé sur la *nature non-optimale* de la meilleure solution trouvée jusqu'alors. Si nous pouvons savoir quelles décisions ont empêché d'affecter une meilleure valeur à la variable objectif (en minimisation, une valeur plus petite que la valeur courante, $I[\Omega]$), alors nous pourrions concevoir des voisinages plus à même d'améliorer la valeur affectée à la variable objectif. Un tel voisinage peut être obtenu en extrayant les décisions en relation avec les valeurs supprimées du domaine de la variable objectif. Ce voisinage est nommé exp-obj, son raisonnement est schématiquement présenté dans la Figure 4.2.

Il est décrit dans l'Algorithme 7. Le point d'entrée est la méthode RELAX_EXP-OBJ.

À chaque fois qu'une nouvelle solution est trouvée (ligne 3), la méthode EXPLAINDOMAIN est appelée (ligne 4). Elle retourne l'ensemble de décisions en relation avec le retrait des valeurs inférieures à $I[\Omega]$ du domaine de la variable objectif. Certaines décisions sont alors supprimées du chemin de décision B_S (lignes 8-13), et le chemin relaxé est appliqué (ligne 14). La manière dont les décisions sont sélectionnées pour être supprimées de B_S est conditionnée par le nombre d'essais déjà effectué. Au départ, les décisions sont sélectionnées pour être supprimées dans l'ordre croissant des valeurs du domaine (lignes 9-10) : les décisions impliquant le retrait de la valeur i du domaine de Ω sont relaxées avant celles impliquant le retrait de la valeur $i+1$. Lorsque toutes les décisions ont été supprimées du chemin de décisions (ligne 11) mais qu'aucune solution n'a pu être trouvée, alors les décisions à supprimer sont choisies aléatoirement (ligne 12). La méthode RANDOM est la même que celle décrite dans l'Algorithme 6, ligne 7.

La méthode EXPLAINDOMAIN (lignes 16-24) fonctionne de la manière suivante : tout d'abord, le nombre de valeurs supprimées du domaine de la variable objectif est calculé (ligne 17). Chacune des valeurs supprimées va alors être expliquée. Pour ce faire, un parcours de ces valeurs, de $\underline{D^i}(\Omega)$ à $I[\Omega]-1$, est effectué (lignes 19-23). La base d'explications est consultée pour accéder aux explications de chaque retrait (appel à la méthode EXPLAINREMOVAL, ligne 20). Chaque retrait de valeur s'explique par la prise d'une ou plusieurs décisions et l'application d'une ou plusieurs contraintes. Ici, on ne s'intéresse qu'aux décisions, elles sont extraites de l'explication calculée (ligne 21) et mémorisées dans Δ_d. Δ_d est un ensemble ordonné de décisions. Pour une valeur donnée, les décisions expliquant son retrait de Ω sont ajoutées dans Δ_d, si elles en étaient absentes. L'évolution des cardinalités

Algorithme 7 Voisinage guidé par le domaine de la variable objectif (dans un contexte de minimisation)

Require: Ω : la variable objectif
Require: k : un entier
Require: Δ_d : l'ensemble ordonné des décisions en relation avec le domaine courant de Ω
Require: I : un tableau d'entiers
1: **procedure** RELAX_EXP-OBJ(S)
2: $B \leftarrow$PATHTO(S) ▷ Extrait le chemin de décision menant à S
3: **if** une nouvelle solution a été trouvée **then**
4: EXPLAINDOMAIN(S, Ω)
5: $k \leftarrow 0$
6: **end if**
7: $k \leftarrow k + 1$
8: $R \leftarrow \emptyset$
9: **if** $k \leq length(I)$ **then**
10: $R \leftarrow \bigcup_{j=1}^{I[k]} \Delta_d[j]$
11: **else**
12: $R \leftarrow$RANDOM(Δ_d) ▷ Sélectionne aléatoirement des décisions de Δ_d
13: **end if**
14: APPLY($B \setminus R$) ▷ Applique B moins R
15: **end procedure**
16: **procedure** EXPLAINDOMAIN(S, Ω)
17: $n \leftarrow (I[\Omega] - \underline{D^i}(\Omega))$ ▷ Calcule le nombre de valeurs retirées de Ω
18: $\Delta_d \leftarrow []$; $I \leftarrow []$;
19: **for** $k \in [1, n]$ **do**
20: $\mathcal{E} \leftarrow$ EXPLAINREMOVAL($D^i(\Omega), \underline{D^i}(\Omega) + k - 1$)
21: $\Delta_d \leftarrow \Delta_d \bigcup$ EXTRACTDECISION($\mathcal{E}$) ▷ Consulte la base d'explications
22: $I[k] \leftarrow |\Delta_d|$
23: **end for**
24: **end procedure**

de Δ_d est également sauvegardée dans I (ligne 22) au fur et à mesure que les valeurs sont expliquées. I est un tableau d'indices. Quand la $k^{\text{ème}}$ valeur retirée de Ω est expliquée, la taille de Δ_d est mémorisée dans I. Grâce à I il est possible de reconstruire l'historique des explications. Les opérations de relaxation (ligne 10) établissent que, tant que k est inférieur à $length(I)$ (ligne 9), seules les décisions en relation avec le retrait de valeurs inférieures à $\underline{D^i}(\Omega) + k$ sont supprimées du chemin de décisions. Ceci est garanti par la manière dont Δ_d et I sont calculés. Dans [54], Jussien et Lhomme montrent que, pour une variable donnée, les explications liées au retrait d'une valeur dépendent, par construction, des explications des valeurs précédemment supprimées. Dans le contexte de notre étude, comme le parcours des valeurs se fait de $\underline{D^i}(\Omega)$ à $I[\Omega] - 1$, cette observation fournit la propriété suivante sur la variable objectif.

Propriété 4.1 *Étant donnée une solution S de coût $I[\Omega] \in [\underline{D^i}(\Omega), \overline{D^i}(\Omega)]$ et le chemin de décision B_S, pour toutes valeurs $v, v' \in [\underline{D^i}(\Omega), I[\Omega][$, $v \leq v'$, et étant donné $\Delta_{\{w\}}$ l'ensemble de décisions expliquant le retrait de la valeur w du domaine de Ω, alors :*

$$\Delta_{\{v\}} \subseteq \Delta_{\{v'\}}$$

La *boucle-pour* de l'Algorithme 7, lignes 19-23, repose sur la Propriété 4.1. Elle construit incrémentalement l'ensemble des décisions candidates à la relaxation en même temps que le tableau d'indices, en parcourant les valeurs supprimées du domaine de Ω, de $\underline{D^i}(\Omega)$ à $I[\Omega] - 1$. Donc, lorsque les décisions liées aux retraits des k premières valeurs de Ω doivent être supprimées du chemin de décision (ligne 10), il suffit de supprimer, de B_S, les $I[k]$ premières décisions de l'ensemble ordonné Δ_d.

Exemple 4.2 (`exp-obj`) *Dans l'exemple 2.8, la variable objectif Ω est instanciée à la valeur 6 dans la solution. Les explications de chaque retrait de valeur du domaine de Ω pendant*

la phase de résolution du problème d'optimisation sont reportées dans le tableau 2.3.

D'un côté, les valeurs inférieures à $I[\Omega]$ s'expliquent à l'aide des décisions δ_2, δ_3 et δ_4. Les retraits des valeurs 0 et 1 du domaine de Ω s'expliquent grâce à l'application de la décision δ_3, au travers de la propagation de la contrainte $C1$ (Tableau 2.2, ligne 3.b). Les retraits des valeurs de 2 à 5 dépendent de l'exécution de la contrainte $C1$: les bornes inférieures des variables x_3, x_5 et x_6 permettent de déduire que Ω ne peut pas prendre de valeurs plus petites que 6 (Tableau 2.2, ligne 4.c). Les décisions δ_2 et δ_3 expliquent l'état courant de la borne inférieure de x_3. La décision δ_4 explique, quant à elle, les valeurs courantes des bornes inférieures de x_5 and x_6.

D'un autre côté, les retraits des valeurs plus grandes que $I[\Omega]$ s'expliquent grâce à δ_4 and δ_5. Les retraits des valeurs 9 et 10 viennent de l'application de δ_4. L'application de δ_5 déclenche le retrait des valeurs 7 et 8. Comme nous nous intéressons à améliorer la valeur de la variable objectif, nous ne retiendrons que les explications liées aux retraits des valeurs inférieures à $I[\Omega] = 6$. Ainsi, δ_5 ne fera pas partie de Δ_d, et l'exécution de la méthode EXPLAINDOMAIN *se termine avec $\Delta_d = \{\delta_3, \delta_2, \delta_4\}$ et $I = \{1, 1, 3, 3, 3, 3\}$. En fait, nous discuterons par la suite d'une manière simple de rendre I plus compact.*

Sur un appel à la méthode RELAX_EXP-OBJ, *δ_1 et δ_5 doivent être maintenues, parce qu'elles n'expliquent le retrait d'aucune valeur inférieure à $I[\Omega]$ du domaine de Ω. Puis, les deux premiers voisinages imposeront δ_3 et relaxeront δ_2 et δ_4. Les quatre suivants relaxeront toutes les décisions de Δ_d. Enfin, tout nouvel appel à la méthode* RELAX_EXP-OBJ *sélectionnera aléatoirement parmi les décisions de Δ_d celles qui seront maintenues avec δ_1 et δ_5, jusqu'à ce qu'une nouvelle solution soit trouvée.*

Dans l'exemple 4.2, δ_3 (respectivement, $\delta_2 \wedge \delta_4$) déclenche le retrait de deux (respectivement, quatre) valeurs consécutives du domaine de Ω. En conséquence, même si Δ_d a une taille adaptée, l'indice 1 apparait deux fois dans I, et l'indice 3 y apparait quatre fois. C'est pourquoi certains voisins sont retournés plusieurs fois par le voisinage. En considérant le retrait d'*intervalles* (*c.-à-d.* un ensemble de valeurs consécutives), au lieu de retrait de valeurs, nous pouvons réduire la taille de I et définir des relaxations du chemin de décisions plus pertinentes. Appliquer à l'exemple précédent 4.2, la boucle considérera deux retraits d'intervalles, $[0, 1]$ et $[2, 5]$, au lieu de six retraits de valeur. L'ensemble ordonné de décisions et le tableau d'indices seront $\Delta_d = \{\delta_3, \delta_2, \delta_4\}$ et $I = \{1, 3\}$. Les deux premiers voisinages seront : $\{\delta_1, \delta_3; \delta_5\}$ et $\{\delta_1, \delta_5\}$, avant de basculer dans la un mode de sélection aléatoire. Nous avons utilisé la gestion par intervalle dans notre implémentation.

Ces voisinages peuvent être vus comme une version inutilement compliquée d'une technique de *round-robin*, dans laquelle les premières décisions du chemin de décisions seraient supprimées en premier. Comme le montre l'exemple 4.2, les décisions liées aux retraits de valeurs (inférieures) du domaine de la variable objectif ne sont pas nécessairement supprimées en premier du chemin de décisions. De plus, les décisions à relaxer ne sont pas sélectionnées aveuglement, et sont directement liées à la variable objectif au travers des explications.

Une approche à base d'explications s'inscrit parfaitement dans une recherche à voisinage large pour traiter de problèmes d'optimisation. Malgré cela, sa mise en œuvre peut paraitre complexe en comparaison avec la simplicité de LNS. Et bien qu'il ne soit pas nécessaire que les voisinages de LNS soient triviaux à implanter, l'instrumentation intrusive d'un solveur pour y intégrer un système à base d'explications peut nuire à la standardisation de notre approche. Plus généralement, il s'agit de la limite principale à la généralisation de techniques à base d'explications. D'autre part, une recherche à voisinage large repose sur sa capacité à produire rapidement beaucoup de voisins. On peut craindre que la complexité des calculs de voisinages reposant sur les explications aille à l'encontre de ce principe primordial. Il est

alors souhaitable que la qualité –et la pertinence– des voisinages proposés compense cet aspect. Avant de valider empiriquement cette hypothèse dans le chapitre 5, nous proposons des axes d'amélioration du système d'explications et des algorithmes présentés dans ce chapitre.

4.3 Informations complémentaires et améliorations

Cette section détaille la méthode de relaxation du chemin de décisions et les techniques adoptées pour améliorer l'efficacité des approches décrites dans les sections 4.1 et 4.2.

4.3.1 Relaxer le chemin de décisions

Dans ce paragraphe, nous décrivons comment le chemin de décisions est effectivement relaxé (Algorithme 6, ligne 7 et Algorithme 7, ligne 14). La méthode APPLY($B \setminus R$) a pour objectif d'appliquer partiellement le chemin de décisions B. Premièrement, toutes les décisions de R sont supprimées du chemin de décisions B. Ensuite, les décisions réfutées doivent être traitées, même si elles n'apparaissent pas dans R. En effet, une décision δ est réfutée quand l'algorithme de recherche clos le sous-arbre qu'elle induit, indiquant que le sous-arbre a été entièrement exploré. Une décision qui est réfutée s'explique par les décisions prises avant elle dans le chemin de décisions. Ainsi, si une décision qui explique la négation d'une autre décision est supprimée, alors il n'y a pas de raisons de maintenir la décision réfutée non plus, elle doit donc être retirée.

Exemple 4.3 (Décisions réfutées) *Étant donné* $B = \{\delta_1, \delta_2, \delta_3, \neg\delta_4\}$ *un chemin de décisions et* $R = \{\delta_2\}$ *l'ensemble des décisions à supprimer de* B*, le chemin relaxé* B' *est égal à* $\{\delta_1, \delta_3\}$*.* $\neg\delta_4$ *est automatiquement retirée parce qu'elle se justifie par* $\delta_1 \wedge \delta_2 \wedge \delta_3$*.*

Plus de détails concernant les explications des décisions réfutées se trouvent dans le dossier d'Habilitation à Diriger des Recherches [51] de Narendra Jussien.

4.3.2 Enregistrement paresseux des explications

Les algorithmes de backtracks dits "intelligents" (Section 2.3.2) accèdent à la base d'explications sur chaque conflit [42, 89, 50]. Nos approches, au contraire, requièrent un accès uniquement lorsque des solutions sont trouvées. En conséquence, il n'est pas pertinent de calculer et de mémoriser les explications au cours de la résolution. Pour limiter la consommation CPU (liée aux calculs des explications) et mémoire (liée à leur stockage), nous avons implémenté une manière *paresseuse* et asynchrone de maintenir la base d'explications, similaire à celle décrite par Gent et autres dans [39]. Les données minimales relatives aux événements générés pendant la résolution (*c.-à-d.* la variable, la modification subie et la cause) sont stockées dans une file tout au long de la résolution. Cette file est rendue *backtrackable* pour ne mémoriser que les informations pertinentes, et oublier celles inutiles au backtrack. À tout moment, on ne mémorise que les événements liés à la branche courante de l'arbre de recherche. Quand une solution est trouvée, la base d'explications doit être consultable, cependant, elle peut être vide ou désynchronisée. Les opérations suivantes sont alors exécutées : les données stockées dans la file sont sorties une à une (en respectant l'ordre chronologique), et les explications sont calculées et stockées dans la base d'explications. Une fois la file vidée, la base d'explications est synchronisée et consultable.

Même si stocker les données dans une file backtrackable se fait à un coût, il est négligeable comparé à celui de maintenir la base d'explications tout au long de la résolution. Les

consommations mémoire et CPU en sont également réduites. Brancher les explications de manière paresseuse et asynchrone sans jamais consulter la base d'explications, et donc sans calculer les explications, ralentit la résolution de moins de 10 % en moyenne.

4.3.3 Expliquer les retraits d'intervalles

La plupart du temps, les explications traitent la réduction d'un domaine comme une séquence de retrait de valeurs. La modification d'une borne inférieure, de i à $k-1$, s'explique par le retrait de chacune des valeurs j comprises entre i et $k-1$, sans prendre en compte la relation qui peut les unir. Un tel comportement devient pathologique dès lors que les variables ont de grands domaines, ce qui est souvent le cas pour la variable objectif. Nous avons d'ailleurs posé l'hypothèse 4.1 selon laquelle le domaine de la variable objectif est, en pratique, représenté par deux entiers, définissant ses bornes. Dans ce cas, il est obligatoire d'expliquer les retraits d'intervalles : cela permet de ne pas calculer ni stocker trop d'informations, et donc sauvegarder de la mémoire et du temps CPU. Notre approche (Section 4.2) adapte la technique décrite par Jussien et Lhomme [54], initialement proposée pour les CSP numériques, aux domaines entiers représentés par des intervalles. Il peut être noté par ailleurs que les *Lazy Clause Generation* solveurs gèrent nativement les retraits d'intervalles [111]. Dans les LCG solveurs, chaque valeur d'un domaine d'une variable entière est représentée par deux variables booléennes $[\![x = v]\!]$ et $[\![x \leq v]\!]$. Par exemple, la variable $[\![x = v]\!]$ est vraie si la variable x est affectée à la valeur v et faux si elle ne peut pas être affectée à v. La gestion mémoire peut également devenir critique pour ces solveurs.

4.3.4 Traiter le cas d'une variable objectif avec un domaine énuméré

Nous avons posé l'hypothèse 4.1 selon laquelle le domaine de la variable objectif est souvent borné en pratique. Or, dans certains cas, il est plus intéressant de représenter explicitement toutes les valeurs de son domaine. Mais alors, les valeurs peuvent ne pas avoir été retirées de manière consécutive du domaine de la variable. Cela remet en cause la Propriétés 4.1 (Section 4.2, et donc le comportement du voisinage `exp-obj`.

Exemple 4.4 (Domaines énumérés) *Le tableau 4.2 reporte les explications associées au domaine énuméré d'une variable objectif Ω. Le chemin de décisions est $B = (\delta_1, \delta_2, \delta_3, \delta_4)$. Comme la variable Ω autorise de faire des trous dans son domaine, c.-à-d. pas seulement modifier ses bornes, l'application de la méthode* RELAX_`EXP-OBJ` *produit le résultat suivant : $\Delta_d = \{\delta_4, \delta_2, \delta_3, \delta_1\}$, $I = \{2, 3, 4, 4\}$. Donc, le premier voisinage relaxera δ_4 and δ_2 qui restaureront les valeurs 0 and 4 dans le domaine de Ω. Le second voisinage relaxera δ_4, δ_2 et δ_3 qui restaureront les valeurs 0, 1, 3 et 4.*

TABLE 4.2 – Explications pour une variable objectif à domaine énuméré.

Variable	{valeur retirée} ← Explications
Ω	$\{0\} \leftarrow (\delta_4 \wedge \delta_2)$,
	$\{1\} \leftarrow \delta_3$,
	$\{2\} \leftarrow \delta_1$,
	$\{3\} \leftarrow (\delta_2 \wedge \delta_3)$,
	$\{4\} \leftarrow \delta_2$

Lorsqu'on considère une variable objectif avec un domaine énuméré, l'exécution de la méthode RELAX_EXP−OBJ peut résulter en une relaxation *plus faible* du chemin de décisions, *c.-à-d.* des valeurs de Ω seront restaurées sans plus respecter l'ordre lexicographique du domaine. Pour autant, cela n'empêche pas d'utiliser ce voisinage pour traiter des variables énumérées.

4.3.5 Reconsidérer le nombre de décisions sélectionnées

Dans la section 4.1, nous avons expliqué comment la sélection aléatoire des décisions à relaxer fonctionne : α décisions sont sélectionnées pour être supprimées, où α est tiré aléatoirement. Cependant, le paramètre α n'est pas tiré à chaque appel de la méthode RANDOM, mais tous les $\theta = minimum(\binom{|\Delta|-1}{\alpha}, 200)$ appels, où Δ est égal à Δ_c ou Δ_d. Cela permet de tester toutes les combinaisons possibles lorsque $\binom{|\Delta|-1}{\alpha}$ est inférieur à 200[1], et de n'en tester qu'un échantillon quand ce nombre est grand –plusieurs voisinages de la même taille sont testés avant de tirer une nouvelle valeur pour α. Nous avons évalué d'autres approches, celle-ci a donné les résultats les plus stables et améliore globalement la résolution.

[1] Cette valeur a été retenue comme limite parce qu'elle permet de trouver un compromis entre énumération complète et échantillonnage, quelles que soient les instances traitées.

5

Expérimentations

L'objectif principal des voisinages `exp-cft` and `exp-obj` est de générer des solutions partielles de bonne qualité, *c.-à-d.* qui orientent l'exploration de l'espace de recherche vers des parties améliorantes et donc rendant plus efficace encore la LNS. Cette section démontre les bénéfices de combiner `exp-obj` et `exp-cft` ensemble.

5.1 Implémentation de la LNS

Dans cette section, nous présentons les détails techniques d'implémentation préalables aux expérimentations.

Recherche à voisinage large guidée par la propagation Indépendant de l'instance traitée mais nécessitant un paramétrage, cette combinaison de trois voisinages a montré son efficacité sur une version modifiée du problème de Car Sequencing [85].

À chaque appel du *voisinage guidé par la propagation*, ou `pgn` (pour "*propagation-guided neighborhood*"), une première variable est sélectionnée aléatoirement pour faire partie de la solution partielle, en lui affectant sa valeur dans la solution précédente. Cette instanciation partielle est propagée au travers du réseau de contraintes et un graphe de dépendance entre les variables est construit : les variables modifiées par propagation sont marquées. Chaque variable marquée non instanciée est placée dans une *liste à priorité*, où les variables sont triées en fonction de l'importance de la réduction que leur domaine a subie. La première variable est ensuite sélectionnée pour faire partie de la solution partielle. La phase de sélection-propagation s'arrête lorsque la somme des logarithmes du domaine des variables est inférieure à une constante K donnée. Cette constante peut être multipliée par un coefficient de correction, nommé epsilon, dont la valeur peut évoluer dynamiquement.

Cet algorithme se décline en deux versions : le voisinage guidé par propagation inverse, ou `repgn` (pour "*reverse propagation-guided neighborhood*") est construit par en sélectionnant une variable pour appartenir à l'instanciation partielle selon une relation de *proximité* liée au volume de propagation qu'elle génère ; le voisinage guidé par la propagation aléatoire, ou `rapgn` (pour "*random propagation-guided neighborhood*") est une version de `pgn` avec

une liste de priorités de taille nulle. Donc, cette première heuristique est, en fait, une application séquentielle de `pgn`, `repgn` et `rapgn`. Nous avons utilisé les paramètres par défaut décrit par Perron et autres dans [85] : la taille de liste est limitée à 10, la constante d'arrêt K a pour valeur 30, et epsilon évolue dynamiquement. Comme nous considérons la LNS comme une *boîte noire* dans notre étude, nous n'avons pas cherché à adapter les paramètres au problème traité, et nous nous sommes concentrés sur l'aspect générique de cette approche. Ce candidat sera nommé *PGLNS*.

Recherche à voisinage large basée sur les explications Nous allons évaluer plusieurs combinaisons des voisinages à base d'explications présentés auparavant. La première combinaison est composé du voisinage `exp-obj` et d'un voisinage aléatoire, nommée `ran`. Ce dernier voisinage apporte de la diversification en générant des solutions partielles nullement liées au problème traité. En effet, dans leur article [85], les auteurs " *obtiennent de meilleurs résultats en intercalant des voisinages purement aléatoires avec des voisinages avancés*". `ran` relaxe ζ variables sélectionnées aléatoirement dans chaque solution partielle générée. Les variables restantes sont, bien entendu, instanciées à leur valeur dans la meilleure solution trouvée jusqu'alors. ζ est initialisé à $\frac{|X|}{3}$ à chaque nouvelle solution, puis sa valeur est augmentée de un tous les 200 appels à ce voisinage. De tels paramètres permettent une bonne diversification. Ce premier candidat est nommé *objLNS*. La seconde combinaison associe le voisinage `exp-cft` et `ran`, est nommé *cftLNS*. Comme `exp-obj` et `exp-cft` exploitent les explications de manières différentes, ils se complètent et le comportement de chacun est renforcé par celui de l'autre. C'est pourquoi, nous évaluons un troisième candidat, nommé *EBLNS*. Il s'agit d'une combinaison de `exp-obj`, `exp-cft` et `ran`. Dans chaque combinaison, les voisinages sont appliqués séquentiellement, jusqu'à obtenir une nouvelle solution.

Fast restarts. Tous les candidats sont évalués avec une stratégie de *fast restart* [83] branchée : elle limite à 30 le nombre d'échecs autorisés avant de générer un nouveau voisin et de redémarrer la recherche. Une telle stratégie est communément associée à la LNS, et a été démontrée comme améliorant globalement son fonctionnement.

Un candidat purement aléatoire, exclusivement composé du voisinage `ran`, facile à implémenter et totalement générique a également été évalué. Cependant, les résultats que ce voisinage donne sont incomparablement moins bons que ceux de PGLNS et EBLNS, c'est pourquoi nous avons choisi de ne pas les reporter ici.

5.1.1 Protocole des évaluations

Les voisinages présentés dans cette section (PGLNS, objLNS, cftLNS et EBLNS) ont été implémentés dans le solveur de contraintes Choco-3.1.0 [90, 91]. Toutes les expérimentations ont été faites sur un Macbook Pro avec un 6-core Intel Xeon cadencé à 2.93Ghz sur MacOS 10.6.8, et Java 1.7. Les résolutions ont été lancées avec une limite de temps de 15 minutes. À cause du caractère aléatoire des voisinages, chaque résolution a été lancée dix fois, les moyennes arithmétiques de l'objectif et l'amplitude[1] sont reportés. L'amplitude donne une indication de la stabilité de chaque approche.

[1] L'amplitude est calculée de la manière suivante : $100 * \frac{maximum - minimum}{moyenne}$

5.1.2 Descriptions des problèmes

Les expérimentations proposées ici sont basées sur dix problèmes et 49 instances. Il s'agit de neuf problèmes d'optimisation extraits du *MiniZinc Challenge* 2012 et 2013, et d'un problème additionnel, le problème de Car Sequencing[2]. Ce dernier problème a été ajouté pour faciliter la comparaison avec PGLNS.

Nous avons gardé les instances pour lesquelles l'algorithme de backtrack classique trouvait au moins une solution dans la limite de 15 minutes, puisque la LNS nécessite une première solution pour être activée. Pour rappel, la première solution est la même pour toutes les résolutions d'un problème donné, quelle que soit l'approche testée.

Parmi les problèmes retenus, on compte cinq problèmes de minimisation : le problème Modified Car Sequencing (`car_cars`), le problème Restaurant Assignment (`fastfood`), le problème League Model (`league_model`), le problème Resource-Constrained Project Scheduling (`rcpsp`) et le problème Vehicle Routing (`vrp`). On compte également cinq problèmes de maximisation : le problème Maximum Profit Subpath (`mario`), le problème Itemset Mining (`pattern_set_mining`), le problème Prize Collecting (`pc`), le problème Ship Scheduling (`ship_schedule`) et le problème Still Life (`still_life`). Les détails de chaque modèle sont détaillés dans le tableau 5.1.

Dans les modèles utilisés dans les expérimentations, aucune contrainte globale n'a été utilisée. En effet, une contrainte globale requiert l'implémentation d'un schéma d'explications spécifique, dont l'évaluation n'est pas l'objectif de cette évaluation. Les contraintes *intégrées*, *c.-à-d.* devant être obligatoirement supportées par le solveur cible, sont quant à elles nativement expliquées dans Choco-3.1.0. Les problèmes suivants, déclarant initialement au moins une contrainte globale, ont été modélisés à l'aide de ces contraintes intégrées : le problème Modified Car Sequencing, le problème League Model, le problème Resource-Constrained Project Scheduling, le problème Maximum Profit Subpath et le problème Itemset Mining. De plus, notons que la moitié des problèmes est définie avec une stratégie statique : `input_order`, qui sélectionne la prochaine variable à instancier suivant un ordre pré-établi statiquement.

[2]Le problème modifié de Car Sequencing (5 instances) est une version du problème de Car Sequencing dans laquelle une configuration sans option est ajoutée, comme décrit par Perron et autres dans [85]. L'objectif est de planifier les voitures qui requièrent cette configuration à la fin, de manière à trouver la solution au problème de satisfaction original.

TABLE 5.1 – Descriptions des problèmes traités.

Problème	Contraintes	Recherche
car_cars	array_int_element, bool2int, int_eq, int_eq_reif, int_lin_eq, int_lin_le	{input_order, indomain_max}
fastfood	int_abs, int_lin_eq, int_lt, int_min, set_in	{input_order, indomain_min}
league_model	array_bool_or, bool2int, bool_le, int_eq_reif, int_le, int_le_reif, int_lin_eq	{first_fail, indomain_max} ∧ {first_fail, indomain_min}
rcpsp	array_bool_and, bool2int, bool_eq_reif, bool_le, int_le_reif, int_lin_le, int_lin_le_reif	{smallest, indomain_min} ∧ {input_order, indomain_min}
vrp	int_le, int_lin_eq, int_lin_le	{first_fail, indomain_min}
mario	array_bool_and, array_bool_or, array_int_element, array_var_bool_element, array_var_int_element, bool2int, bool_eq_reif, bool_le, bool_le_reif, int_eq, int_eq_reif, int_lin_eq, int_lt_reif, int_min, int_ne, int_ne_reif	{first_fail, indomain_min} ∧ {input_order, indomain_max}
pattern_set_mining	bool2int, bool_eq, int_lin_eq, int_lin_le_reif	{input_order, indomain_max}
pc	array_bool_and, array_int_element, array_var_int_element, bool2int, bool_le, int_eq, int_eq_reif, int_lin_eq, int_lin_le, int_lt_reif	{largest, indomain_max} ∧ {largest, indomain_max} ∧ {input_order, indomain_max}
ship_schedule	array_bool_and, array_bool_or, array_int_element, bool2int, bool_le, int_eq, int_eq_reif, int_le_reif, int_lin_eq, int_lin_le, int_lin_le_reif, int_lt_reif, int_times	{input_order, indomain_max} ∧ {input_order, indomain_min}
still_life	array_bool_and, bool_le, int_eq, int_eq_reif, int_le_reif, int_lin_eq, int_lin_le, int_lin_le_reif, int_lin_ne_reif, int_times	{input_order, indomain_max}

5.2 Évaluation de objLNS, cftLNS et EBLNS

L'objectif principal de cette partie est double. Premièrement, les voisinages totalement génériques, basés sur les explications, sont mis en œuvre. Deuxièmement, nous montrons que ces voisinages permettent de produire des voisins qui peuvent être facilement étendus à des solutions, et donc améliorer la résolution des problèmes d'optimisation. Dans cette section, nous comparons les candidats basés sur les explications, objLNS, cftLNS et EBLNS, avec celui basé sur la propagation, PGLNS.

Pour présenter les résultats, nous les compilons dans des tableaux qui reportent, pour chacun des candidats évaluées, les moyennes arithmétiques de l'objectif et son amplitude. Une valeur en gras souligne la meilleure valeur trouvée pour la variable objectif, des valeurs en italique mettent en perspective les égalités. Nous utilisons également des graphiques montrant par combien la valeur de la variable objectif est multipliée en appliquant une approche plutôt qu'une autre. L'axe horizontal représente les instances traitées, triées en fonction de la différence entre la solution trouvée avec la première approche h_1 et celle trouvée avec la seconde h_2, par ordre croissant. L'axe vertical reporte le coefficient multiplicateur $\rho(h_i, h_j) = max(\frac{h_i}{h_j}, 1)$. Un point dans la partie gauche du graphique (région verte) reporte $\rho(h_1, h_2)$ pour une instance mieux résolue avec h_1, il mesure alors la dégradation liée à l'utilisation de h_2 plutôt que h_1. Un point dans la partie droite du graphique (région bleue) reporte $\rho(h_2, h_1)$ pour une instance mieux résolue avec h_2, il mesure alors l'amélioration liée à l'utilisation de h_2 plutôt que h_1. Le graphique reporte également A, une approximation de la surface[3] des gains et pertes. Plus la surface verte (respectivement, bleue) est grande et plus l'amélioration liée à h_1 (respectivement, h_2) est importante.

5.2.1 Évaluation comparative de PGLNS et objLNS

Les résultats obtenus avec les approches PGLNS et objLNS sur les dix problèmes sont reportés dans le tableau 5.2. PGLNS et objLNS se partagent les résultats de la manière suivante : pour 19 des 49 instances, PGLNS est la meilleure approche, alors que objLNS est plus efficient pour 26 instances. Dans environ un tiers de cas (6 sur 19 pour PGLNS, 8 sur 26 pour objLNS), l'amplitude est nulle, ce qui signifie que les dix résolutions d'une instance effectuée avec une approche ont toutes abouties à la même dernière solution. On compte quatre instances où les deux approches sont équivalentes.

D'un côté, objLNS produit les meilleures solutions pour les problèmes Restaurant Assignment (`fastfood`), League Model (`league_model`), Prize Collecting (`pc`) et Still Life (`still_life`). Concernant les problèmes Restaurant Assignment (`fastfood`) et Prize Collecting (`pc`), objLNS est plus stable que PGLNS, ce qui n'est pas le cas pour les instances des autres problèmes. Ce candidat est également approprié au traitement du problème Vehicle Routing (`vrp`), mais les deux candidats sont relativement instables sur ces instances. On peut émettre l'hypothèse que les voisinages aléatoires jouent un rôle important dans la résolution des instances de ce problème. De l'autre côté, PGLNS est la meilleure approche pour résoudre le problème Resource-Constrained Project Scheduling (`rcpsp`), il est également plus stable que objLNS sur ces instances. Ce candidat est également approprié au traitement du problème Modified Car Sequencing (`car_cars`), Itemset Mining (`pattern_set_mining`) et Ship Scheduling (`ship_schedule`), où il est aussi plus stable. De manière générale, PGLNS est en moyenne plus stable (16.4 %) que objLNS (24.54 %).

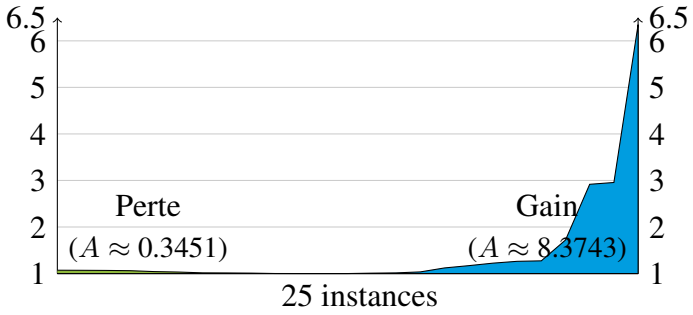
[3] L'approximation de la surface est calculée par la méthode des trapèzes [2].

TABLE 5.2 – Évaluation comparative entre PGLNS et objLNS.

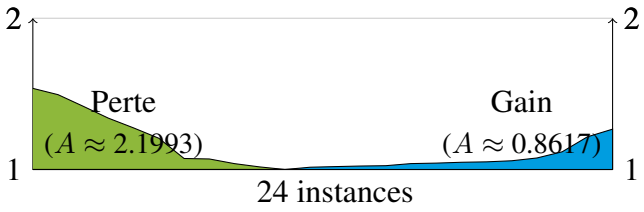
Instance	PGLNS obj	PGLNS amp (%)	objLNS obj	objLNS amp (%)
Problèmes de minimisation				
cars_cars_4_72.fzn	**109.3**	5.49	116.9	4.28
cars_cars_16_81.fzn	121.2	2.48	**119.5**	6.69
cars_cars_26_82.fzn	**127.5**	5.49	136.6	6.59
cars_cars_41_66.fzn	**108.8**	1.84	109.8	5.46
cars_cars_90_05.fzn	**300.8**	11.97	319.3	2.82
fastfood_ff3.fzn	3883.4	90.64	**1331**	0.45
fastfood_ff58.fzn	8882.7	101.78	**1399.2**	10.58
fastfood_ff59.fzn	871	0.00	**294.6**	32.59
fastfood_ff61.fzn	221.7	24.81	**189.6**	29.54
fastfood_ff63.fzn	146.9	40.16	**120.1**	47.46
league_model20-3-5.fzn	*49984*	0.00	*49984*	0.00
league_model30-4-6.fzn	*79973*	0.00	*79973*	0.00
league_model50-4-4.fzn	*99971*	0.00	*99971*	0.00
league_model55-3-12.fzn	139949	35.73	**109949.7**	0.00
league_model90-18-20.fzn	1922916	7.28	**1117921**	39.36
league_model100-21-12.fzn	3139911.9	0.00	**2486918.6**	85.65
rcpsp_11.fzn	**81**	3.70	83.6	2.39
rcpsp_12.fzn	**38.3**	2.61	39.9	5.01
rcpsp_13.fzn	**78**	0.00	79	0.00
rcpsp_14.fzn	**140.9**	0.71	141	0.00
vrp_A-n38-k5.vrp.fzn	2341.6	22.16	**2322.2**	18.30
vrp_A-n62-k8.vrp.fzn	6318.1	6.66	**6104**	29.26
vrp_B-n51-k7.vrp.fzn	**4291.8**	15.87	4355.8	19.74
vrp_P-n20-k2.vrp.fzn	402.1	41.28	**358.9**	21.73
vrp_P-n60-k15.vrp.fzn	**2271.9**	12.28	2424	20.30
Problèmes de maximisation				
mario_mario_easy_2.fzn	*628*	0.00	*628*	0.00
mario_mario_easy_4.fzn	506	8.70	**517.6**	8.50
mario_mario_n_medium_2.fzn	719.9	31.95	**774.7**	22.85
mario_mario_n_medium_4.fzn	**576.6**	52.90	555.1	37.29
mario_mario_t_hard_1.fzn	**4783**	0.00	3199.9	148.00
pattern_set_mining_k1_ionosphere.fzn	**16.9**	130.18	11	100.00
pattern_set_mining_k2_audiology.fzn	**53.6**	1.87	52.7	18.98
pattern_set_mining_k2_german-credit.fzn	3	0.00	**3.8**	131.58
pattern_set_mining_k2_segment.fzn	**11.8**	8.47	11	0.00
pc_25-5-5-9.fzn	62.8	3.18	**64**	0.00
pc_28-4-7-4.fzn	47.7	12.58	**58**	0.00
pc_30-5-6-7.fzn	60	0.00	**60.9**	6.57
pc_32-4-8-0.fzn	104.7	14.33	**109.7**	8.20
pc_32-4-8-2.fzn	84.9	15.31	**89.8**	6.68
ship-schedule.cp_5Ships.fzn	483645.5	0.01	**452157**	19.40
ship-schedule.cp_6ShipsMixed.fzn	**300185**	4.86	248822.5	43.45
ship-schedule.cp_7ShipsMixed.fzn	**396137**	20.40	279396	89.34
ship-schedule.cp_7ShipsMixedUnconst.fzn	**364141.5**	21.01	285652.5	87.13
ship-schedule.cp_8ShipsUnconst.fzn	**782516**	20.51	584308.5	48.27
still-life_09.fzn	37.6	13.30	**42**	0.00
still-life_10.fzn	49.8	4.02	**51.9**	5.78
still-life_11.fzn	59	0.00	**61.3**	1.63
still-life_12.fzn	63.3	4.74	**66.5**	21.05
still-life_13.fzn	79.7	2.51	**81.7**	9.79

Globalement, chaque approche semble adaptée à certaines classes de problèmes, et objLNS apparait comme étant une approche à plus large spectre de résolution. Pour autant, il est dur de qualifier ce qu'apporte une approche par rapport à l'autre à la seule lecture du tableau. C'est pourquoi nous présentons cette information de manière visuelle sous la forme d'un graphique dans la figure 5.1.

FIGURE 5.1 – Coefficient multiplicateur entre PGLNS et objLNS, par instance.



(a) Problèmes de minimisation.



(b) Problèmes de maximisation.

Concernant les problèmes de minimisation (figure 5.1(a)), l'approche basée sur objLNS est clairement plus intéressante : la surface de la région bleue ($A \approx 8.3743$) est bien plus importante que celle de la région verte ($A \approx 0.3451$). Cela indique que objLNS permet de trouver de bien meilleures solutions que PGLNS, jusqu'à un coefficient de 6.34 pour une instance. Un tel écart s'explique simplement : les instances du problèmes Restaurant Assignment (`fastfood`) sont bien mieux résolues avec objLNS qu'avec PGLNS. Pour les autres instances, le gain est moins net.

Concernant les problèmes de maximisation (figure 5.1(b)), les conclusions sont renversées. L'apport de PGLNS ($A \approx 2.1993$) est plus important que celui de objLNS ($A \approx 0.8617$). L'écart, moins important qu'en minimisation, s'explique par les résultats en faveur de PGLNS sur deux classes de problème : Ship Scheduling (`ship_schedule`) et Itemset Mining (`pattern_set_mining`).

En terme de gain, objLNS semble plus intéressante en moyenne, sa contribution à l'obtention de solution de bonne qualité est clairement visible en ce qui concerne les problèmes de minimisation. Pour les problèmes de maximisation, le gain est plus faible que celui apporté par PGLNS mais il est compensé par le nombre d'instances mieux traité (15 instances sur 24).

5.2.2 Évaluation comparative de PGLNS et cftLNS

Les résultats obtenus avec PGLNS et cftLNS sur les 49 instances sont reportés dans le tableau 5.3. Comparativement à PGLNS, cftLNS traite mieux 29 instances sur les 49, alors

TABLE 5.3 – Évaluation comparative entre PGLNS et cftLNS.

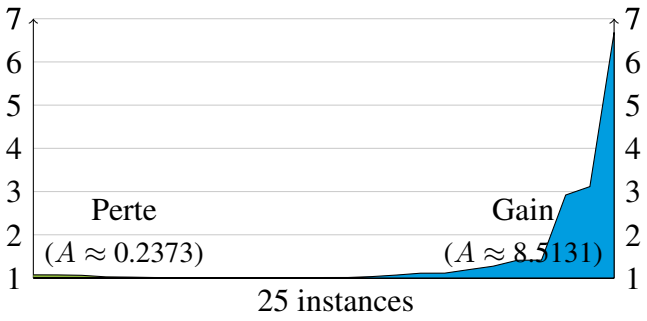
Instance	PGLNS obj	PGLNS amp (%)	cftLNS obj	cftLNS amp (%)
Problèmes de minimisation				
cars_cars_4_72.fzn	**109.3**	5.49	117.1	4.27
cars_cars_16_81.fzn	121.2	2.48	**119.9**	6.67
cars_cars_26_82.fzn	**127.5**	5.49	136.8	5.85
cars_cars_41_66.fzn	**108.8**	1.84	109.8	5.46
cars_cars_90_05.fzn	**300.8**	11.97	319.5	2.50
fastfood_ff3.fzn	3883.4	90.64	**1330**	0.00
fastfood_ff58.fzn	8882.7	101.78	**1329.2**	16.48
fastfood_ff59.fzn	871	0.00	**279.6**	16.81
fastfood_ff61.fzn	221.7	24.81	**157.2**	18.45
fastfood_ff63.fzn	146.9	40.16	**103.9**	0.96
league_model20-3-5.fzn	*49984*	0.00	*49984*	0.00
league_model30-4-6.fzn	*79973*	0.00	*79973*	0.00
league_model50-4-4.fzn	*99971*	0.00	*99971*	0.00
league_model55-3-12.fzn	139949	35.73	**109949.9**	0.00
league_model90-18-20.fzn	1922916	7.28	**1730925.5**	26.58
league_model100-21-12.fzn	3139911.9	0.00	**3103922.1**	2.58
rcpsp_11.fzn	**81**	3.70	81.9	1.22
rcpsp_12.fzn	**38.3**	2.61	39.3	5.09
rcpsp_13.fzn	*78*	0.00	*78*	0.00
rcpsp_14.fzn	**140.9**	0.71	141	0.00
vrp_A-n38-k5.vrp.fzn	2341.6	22.16	**2106.2**	44.44
vrp_A-n62-k8.vrp.fzn	6318.1	6.66	**6117.2**	16.30
vrp_B-n51-k7.vrp.fzn	4291.8	15.87	**4019.1**	29.83
vrp_P-n20-k2.vrp.fzn	402.1	41.28	**336.8**	23.46
vrp_P-n60-k15.vrp.fzn	**2271.9**	12.28	2317.8	29.99
Problèmes de maximisation				
mario_mario_easy_2.fzn	*628*	0.00	*628*	0.00
mario_mario_easy_4.fzn	506	8.70	**510.1**	8.63
mario_mario_n_medium_2.fzn	719.9	31.95	**889**	25.20
mario_mario_n_medium_4.fzn	576.6	52.90	**723.9**	28.60
mario_mario_t_hard_1.fzn	**4783**	0.00	3039.4	157.37
pattern_set_mining_k1_ionosphere.fzn	16.9	130.18	**26.6**	150.38
pattern_set_mining_k2_audiology.fzn	53.6	1.87	**54**	0.00
pattern_set_mining_k2_german-credit.fzn	3	0.00	**4.9**	81.63
pattern_set_mining_k2_segment.fzn	**11.8**	8.47	11	0.00
pc_25-5-5-9.fzn	62.8	3.18	**64.2**	1.56
pc_28-4-7-4.fzn	47.7	12.58	**57.2**	13.99
pc_30-5-6-7.fzn	60	0.00	**62.7**	6.38
pc_32-4-8-0.fzn	104.7	14.33	**111.2**	8.09
pc_32-4-8-2.fzn	84.9	15.31	**91.6**	4.37
ship-schedule.cp_5Ships.fzn	**483645.5**	0.01	483602.5	0.02
ship-schedule.cp_6ShipsMixed.fzn	**300185**	4.86	251359.5	68.71
ship-schedule.cp_7ShipsMixed.fzn	**396137**	20.40	302325.5	10.34
ship-schedule.cp_7ShipsMixedUnconst.fzn	**364141.5**	21.01	318492.5	23.42
ship-schedule.cp_8ShipsUnconst.fzn	**782516**	20.51	470266.5	22.39
still-life_09.fzn	37.6	13.30	**42**	0.00
still-life_10.fzn	49.8	4.02	**53.6**	3.73
still-life_11.fzn	59	0.00	**61.2**	1.63
still-life_12.fzn	63.3	4.74	**75.1**	2.66
still-life_13.fzn	79.7	2.51	**87.2**	3.44

que PGLNS n'est plus efficace que dans 15 cas. On compte six égalités. Dans environ un tiers de cas (5 pour PGLNS, 8 pour cftLNS), l'amplitude vaut 0, indiquant une stabilité optimale des approches.

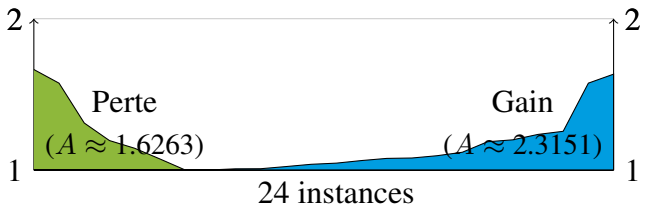
En terme de répartition de classe de problèmes, les résultats sont comparables à ceux obtenus avec objLNS : les heuristiques à base d'explications sont adaptées aux problèmes Restaurant Assignment (`fastfood`), League Model (`league_model`), Prize Collecting (`pc`) et Still Life (`still_life`). Seulement, cftLNS apporte plus de stabilité que objLNS. D'autre part, cftLNS est plus adapté au traitement des problèmes Vehicle Routing (`vrp`), Maximum Profit Subpath (`mario`) et Itemset Mining (`pattern_set_mining`), par rapport à l'approche PGLNS. Il est, par contre, moins évident de statuer quant à sa stabilité, on observe de grands écarts, notamment pour le problème Itemset Mining (`pattern_set_mining`). PGLNS reste l'approche la plus efficace pour résoudre les problèmes Ship Scheduling (`ship_schedule`) et Modified Car Sequencing (`car_cars`). Concernant ce dernier problème, on remarque que PGLNS est battu sur une instance. De manière générale, PGLNS est en moyenne légèrement plus stable que cftLNS, 16.4 % pour PGLNS contre 17.94 % pour cftLNS.

La représentation visuelle du gain apporté par cftLNS par rapport à PGLNS est présenté dans la figure 5.2. Concernant les problèmes de minimisation (figure 5.2(a)), on observe

FIGURE 5.2 – Coefficient multiplicateur entre PGLNS et cftLNS, par instance.



(a) Problèmes de minimisation.



(b) Problèmes de maximisation.

le même comportement pour cftLNS qu'avec objLNS : la région bleue ($A \approx 8.5131$) occupe une plus grande surface que la région verte ($A \approx 0.2373$). Là encore, les instances du problème Restaurant Assignment (`fastfood`) sont bien mieux résolues avec l'approche cftLNS, et sont visibles sous la forme d'un pic à droite sur le graphique. Concernant les problèmes de maximisation (figure 5.2(b)), la tendance est maintenue : cftLNS semble plus efficace. D'une part, la surface de la région bleue ($A \approx 2.3151$) est plus importante que celle de la région verte ($A \approx 1.6263$), ce qui indique bien que cftLNS construit des solutions de meilleure qualité, mais en plus il traite plus efficacement un nombre plus grand d'instances (16 en faveur de cftLNS, 7 en faveur de PGLNS). Globalement, les résultats sont au crédit

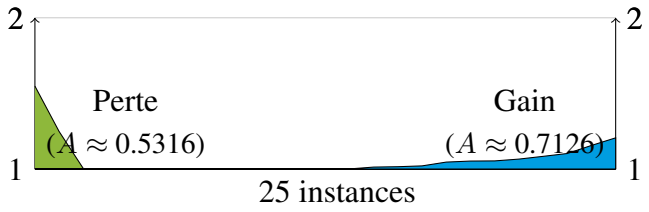
de cftLNS, en terme de nombre d'instances mieux résolues, mais surtout en terme de gain net par rapport à PGLNS.

5.2.3 Évaluation comparative de cftLNS et objLNS

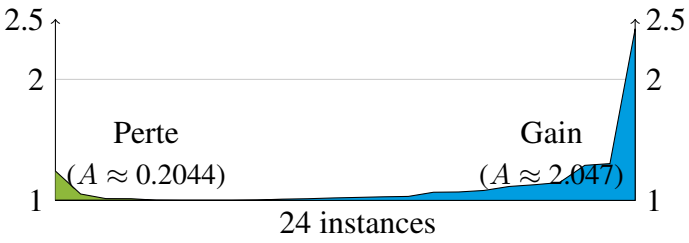
Les résultats de cftLNS et objLNS, comparativement à PGLNS, semblent très similaires. Nous comparons donc ces deux candidats, et reportons les données dans le tableau 5.4. En terme de répartition, cftLNS est plus efficient : il résout mieux 28 instances, contre 13 pour objLNS, et huit égalités. De plus, il semble plus stable en moyenne (17.94 % pour cftLNS contre 24.54 % pour objLNS). Pour autant, objLNS traite mieux les instances des problèmes Modified Car Sequencing (`car_cars`) et League Model (`league_model`). De son côté, cftLNS est plus approprié au traitement des problèmes Restaurant Assignment (`fastfood`), Resource-Constrained Project Scheduling (`rcpsp`), et, dans une moindre mesure, Vehicle Routing (`vrp`), Itemset Mining (`pattern_set_mining`), Prize Collecting (`pc`), Ship Scheduling (`ship_schedule`) et Still Life (`still_life`). Seules les instances du problème Maximum Profit Subpath ne semblent pas moins bien traitées par une des deux approches.

Toutefois, il est intéressant de pondérer cette large victoire de cftLNS par la mesure des gains qu'elle peut apporter, en comparaison avec objLNS. Les résultats sont reportés dans la figure 5.3. Concernant les problèmes de minimisation, les gains des deux candidats

FIGURE 5.3 – Coefficient multiplicateur entre cftLNS et objLNS, par instance.



(a) Problèmes de minimisation.



(b) Problèmes de maximisation.

sont comparables en terme de surface : la région verte occupe un espace ($A \approx 0.5316$) légèrement plus petit que celui occupé par la région bleue ($A \approx 0.7126$), mais cette dernière est plus large que haute. Cela confirme que cftLNS résout mieux plus d'instances, mais que les gains obtenus sont assez faible. En revanche, l'apport de objLNS est plus important mais ne concerne que peu d'instances. Concernant les problèmes de maximisation, l'analyse est sans appel, il est clairement plus rentable d'utiliser cftLNS en lieu et place d'objLNS. Comme chacune des approches est nettement adaptée à certaines classes de problèmes, nous pouvons espérer qu'en les combinant nous bénéficierons des avantages de chacune d'entre elles. C'est l'objet de l'étude suivante.

TABLE 5.4 – Évaluation comparative entre objLNS et cftLNS.

Instance	objLNS obj	objLNS amp (%)	cftLNS obj	cftLNS amp (%)
Problèmes de minimisation				
cars_cars_4_72.fzn	**116.9**	4.28	117.1	4.27
cars_cars_16_81.fzn	**119.5**	6.69	119.9	6.67
cars_cars_26_82.fzn	**136.6**	6.59	136.8	5.85
cars_cars_41_66.fzn	*109.8*	5.46	*109.8*	5.46
cars_cars_90_05.fzn	**319.3**	2.82	319.5	2.50
fastfood_ff3.fzn	1331	0.45	**1330**	0.00
fastfood_ff58.fzn	1399.2	10.58	**1329.2**	16.48
fastfood_ff59.fzn	294.6	32.59	**279.6**	16.81
fastfood_ff61.fzn	189.6	29.54	**157.2**	18.45
fastfood_ff63.fzn	120.1	47.46	**103.9**	0.96
league_model20-3-5.fzn	*49984*	0.00	*49984*	0.00
league_model30-4-6.fzn	*79973*	0.00	*79973*	0.00
league_model50-4-4.fzn	*99971*	0.00	*99971*	0.00
league_model55-3-12.fzn	**109949.7**	0.00	109949.9	0.00
league_model90-18-20.fzn	**1117921**	39.36	1730925.5	26.58
league_model100-21-12.fzn	**2486918.6**	85.65	3103922.1	2.58
rcpsp_11.fzn	83.6	2.39	**81.9**	1.22
rcpsp_12.fzn	39.9	5.01	**39.3**	5.09
rcpsp_13.fzn	79	0.00	**78**	0.00
rcpsp_14.fzn	*141*	0.00	*141*	0.00
vrp_A-n38-k5.vrp.fzn	2322.2	18.30	**2106.2**	44.44
vrp_A-n62-k8.vrp.fzn	**6104**	29.26	6117.2	16.30
vrp_B-n51-k7.vrp.fzn	4355.8	19.74	**4019.1**	29.83
vrp_P-n20-k2.vrp.fzn	358.9	21.73	**336.8**	23.46
vrp_P-n60-k15.vrp.fzn	2424	20.30	**2317.8**	29.99
Problèmes de maximisation				
mario_mario_easy_2.fzn	*628*	0.00	*628*	0.00
mario_mario_easy_4.fzn	**517.6**	8.50	510.1	8.63
mario_mario_n_medium_2.fzn	774.7	22.85	**889**	25.20
mario_mario_n_medium_4.fzn	555.1	37.29	**723.9**	28.60
mario_mario_t_hard_1.fzn	**3199.9**	148.00	3039.4	157.37
pattern_set_mining_k1_ionosphere.fzn	11	100.00	**26.6**	150.38
pattern_set_mining_k2_audiology.fzn	52.7	18.98	**54**	0.00
pattern_set_mining_k2_german-credit.fzn	3.8	131.58	**4.9**	81.63
pattern_set_mining_k2_segment.fzn	*11*	0.00	*11*	0.00
pc_25-5-5-9.fzn	64	0.00	**64.2**	1.56
pc_28-4-7-4.fzn	**58**	0.00	57.2	13.99
pc_30-5-6-7.fzn	60.9	6.57	**62.7**	6.38
pc_32-4-8-0.fzn	109.7	8.20	**111.2**	8.09
pc_32-4-8-2.fzn	89.8	6.68	**91.6**	4.37
ship-schedule.cp_5Ships.fzn	452157	19.40	**483602.5**	0.02
ship-schedule.cp_6ShipsMixed.fzn	248822.5	43.45	**251359.5**	68.71
ship-schedule.cp_7ShipsMixed.fzn	279396	89.34	**302325.5**	10.34
ship-schedule.cp_7ShipsMixedUnconst.fzn	285652.5	87.13	**318492.5**	23.42
ship-schedule.cp_8ShipsUnconst.fzn	**584308.5**	48.27	470266.5	22.39
still-life_09.fzn	*42*	0.00	*42*	0.00
still-life_10.fzn	51.9	5.78	**53.6**	3.73
still-life_11.fzn	**61.3**	1.63	61.2	1.63
still-life_12.fzn	66.5	21.05	**75.1**	2.66
still-life_13.fzn	81.7	9.79	**87.2**	3.44

5.2.4 Évaluation comparative de EBLNS et PGLNS

L'une des forces de LNS est la possibilité qu'elle offre de facilement combiner plusieurs voisinages ensemble. Nous allons maintenant comparer les voisinages à base d'explications avec l'approche à base de propagation. Les résultats sont reportés dans le tableau 5.5. PGLNS et EBLNS se partagent les instances presque équitablement : dans 22 des 49 instances, PGLNS est la meilleure approche, alors que EBLNS est la meilleure approche pour 24 instances. Dans environ un tiers des cas (7 sur 22 pour PGLNS, 7 sur 24 pour EBLNS), l'amplitude vaut 0, ce qui signifie que toutes les résolutions aboutissent à la même solution. On compte seulement trois instances où les deux approches trouvent une dernière solution de même valeur.

D'un côté, EBLNS produit les meilleurs résultats pour le problème Still Life Problem (`still_life`), il est plus stable que PGLNS sur ces instances. Cette combinaison est également plus appropriée à la résolution du problème Vehicle Routing (`vrp`), du problème Itemset Mining (`pattern_set_minning`) et du problème Price Collecting (`pc`). Cependant, les deux candidats sont relativement instables sur ces instances, en particulier, en ce qui concerne le problème Itemset Mining, où les plus grandes amplitudes sont observées. Sur le problème Resource-Constrained Project Scheduling (`rcpsp`), les résultats sont très comparables, même s'ils sont légèrement en faveur de PGLNS, et les deux approches sont très stables. Sur le problème Restaurant Assignment (`fast_food`) et le problème League Model (`league_model`), EBLNS trouve des solutions équivalentes ou meilleures dans plus de cas (7 sur 11), et tend à être plus stable en moyenne (27.3 % pour PGLNS, 19.5 % pour EBLNS).

De l'autre côté, PGLNS trouve les meilleurs résultats pour le problème Ship Scheduling (`ship_schedule`), et est plus stable que EBLNS, sur ces instances. PGLNS est également plus approprié à la résolution du problème Maximum Profit Subpath (`mario`) et du problème Modified Car Sequencing (`car_cars`), le problème pour lequel il a été conçu initialement. Sur ces instances, la stabilité, une fois de plus, est en faveur de PGLNS. D'ailleurs EBLNS n'est pas du tout stable pour les instances du problème Maximum Profit Subpath.

De manière générale, chaque candidat semble plus adapté à la résolution d'une certaine catégorie de problèmes. Malgré cela, cette considération est à mettre en parallèle de l'observation de la stabilité : elle varie d'une instance à l'autre pour une même classe de problèmes. Par contre, PGLNS est en moyenne bien plus stable que EBLNS, 16.4 % pour PGLNS contre 28.50 % pour EBLNS. À ce stade, il est difficile de conclure sur la qualité des solutions partielles produites par les deux approches. EBLNS souffre d'un ralentissement dû aux explications, même si celles-ci permettent de construire de bonnes solutions partielles.

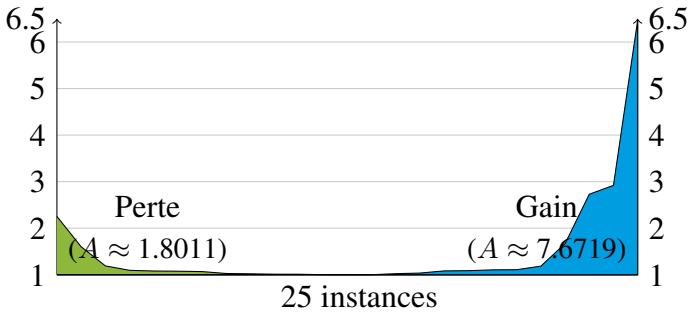
Nous allons, à présent, mesurer le gain d'utiliser EBLNS plutôt que PGLNS sur les instances traitées. Les résultats sont représentés visuellement dans la figure 5.6. Concernant les problèmes de minimisation, les succès de chaque approche sont équitablement répartis, mais le gain lié à l'utilisation de EBLNS à la place de PGLNS est remarquable. La surface de la région bleue ($A \approx 7.6719$) est clairement plus grande que celle de la région verte ($A \approx 1.8011$). EBLNS améliore majoritairement les solutions, et les dégrade dans une moindre mesure. Concernant les problèmes de maximisation, le gain est moins marqué, mais légèrement en faveur de EBLNS. Les surfaces sont comparables, mais EBLNS traite mieux plus d'instances, parfois de peu, que PGLNS : la surface bleue ($A \approx 3.6728$) est plus large que haute.

En ce qui concerne nos attentes premières, c'est à dire tirer partie des avantages des voisinages `exp-cft` et `exp-obj` en les combinant, les résultats sont partagés. D'une part, on observe une atténuation du gain par rapport à PGLNS sur les problèmes de minimisation : la

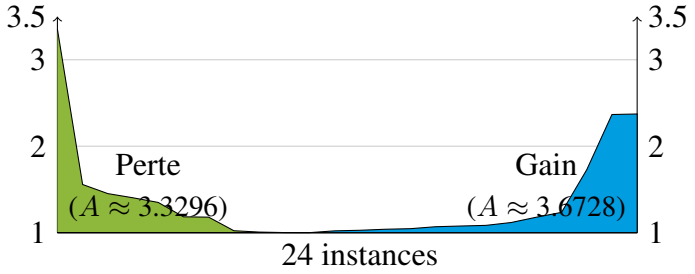
TABLE 5.5 – Évaluation comparative entre PGLNS et EBLNS.

Instance	PGLNS obj	PGLNS amp (%)	EBLNS obj	EBLNS amp (%)
Problèmes de minimisation				
cars_cars_4_72	**109.3**	5.49	118	2.54
cars_cars_16_81	121.2	2.48	**117.1**	11.10
cars_cars_26_82	**127.5**	5.49	139.5	3.58
cars_cars_41_66	**108.8**	1.84	110	0
cars_cars_90_05	**300.8**	11.97	321	0
fastfood_ff3	3883.4	90.64	**1330**	0
fastfood_ff58	8882.7	101.78	**1373**	0
fastfood_ff59	871	0	**319.2**	21.30
fastfood_ff61	**221.7**	24.81	355	0
fastfood_ff63	**146.9**	40.16	331	0
league_model20-3-5	*49984*	0	*49984*	0
league_model30-4-6	**79973**	0	85974	23.26
league_model50-4-4	**99971**	0	118971.7	33.62
league_model55-3-12	139949	35.73	**136950.5**	29.21
league_model90-18-20	1922916	7.28	**1152924.8**	26.89
league_model100-21-12	3139911.9	0	**2651923.1**	80.70
rcpsp_11	**81**	3.70	81.8	1.22
rcpsp_12	**38.3**	2.61	39.3	2.54
rcpsp_13	**78**	0	**78**	0
rcpsp_14	**140.9**	0.71	141	0
vrp_A-n38-k5.vrp	2341.6	22.16	**2114**	36.94
vrp_A-n62-k8.vrp	6318.1	6.66	**5840.4**	27.87
vrp_B-n51-k7.vrp	4291.8	15.87	**3950.8**	30.42
vrp_P-n20-k2.vrp	402.1	41.28	**364.2**	17.57
vrp_P-n60-k15.vrp	**2271.9**	12.28	2316.3	24.87
Problèmes de maximisation				
mario_mario_easy_2	*628*	0	*628*	0
mario_mario_easy_4	**506**	8.70	502.7	3.38
mario_mario_n_medium_2	**719.9**	31.95	610.5	125.31
mario_mario_n_medium_4	576.6	52.90	**711.9**	30.90
mario_mario_t_hard_1	**4783**	0	1426.1	282.87
pattern_set_mining_k1_ionosphere	16.9	130.18	**29.2**	89.04
pattern_set_mining_k2_audiology	**53.6**	1.87	38.2	2.62
pattern_set_mining_k2_german-credit	3	0	**7.1**	169.01
pattern_set_mining_k2_segment	11.8	8.47	**28**	0
pc_25-5-5-9	62.8	3.18	**64.1**	3.12
pc_28-4-7-4	47.7	12.58	**49**	46.94
pc_30-5-6-7	**60**	0	58.7	13.63
pc_32-4-8-0	104.7	14.33	**108.8**	10.11
pc_32-4-8-2	84.9	15.31	**91.4**	6.56
ship-schedule.cp_5Ships	**483645.5**	0.01	483239	0.24
ship-schedule.cp_6ShipsMixed	**300185**	4.86	253749	41.92
ship-schedule.cp_7ShipsMixed	**396137**	20.40	254441.5	102.45
ship-schedule.cp_7ShipsMixedUnconst	**364141.5**	21.01	269851	41.39
ship-schedule.cp_8ShipsUnconst	**782516**	20.51	539091	35.92
still-life_09	37.6	13.30	**42**	0
still-life_10	49.8	4.02	**53.2**	3.76
still-life_11	59	0	**61.7**	1.62
still-life_12	63.3	4.74	**74.6**	5.36
still-life_13	79.7	2.51	**86.4**	6.94

FIGURE 5.4 – Coefficient multiplicateur entre PGLNS et EBLNS, par instance.



(a) Problèmes de minimisation.



(b) Problèmes de maximisation.

surface des gains apportés par EBLNS par rapport à PGLNS sur ces problèmes est plus petite que celles précédemment observées avec cftLNS et objLNS. On observe même une perte légèrement plus importante. En ce qui concerne les problèmes de maximisation, on observe toujours cette augmentation de la perte, par contre, on dénote une amélioration des gains. On peut en conclure que la portion de voisinages expliqués est peut être trop importante dans cette combinaison (deux voisinages sur les trois) ce qui se traduit par un ralentissement du fonctionnement de LNS et empêche de progresser rapidement dans l'obtention de solutions de meilleure qualité.

Par contre, là encore, il y a, proportionnellement, peu d'instances équivalentes : quelle que soit la classe des problèmes, les résultats sont bien marqués. On peut alors parier sur une bonne compatibilité entre les voisinages de PGLNS et EBLNS. Il est donc tout à fait naturel de les combiner pour gommer leurs défauts et renforcer la stabilité globale. L'évaluation de cette combinaison est l'objet de la section suivante.

5.3 Combiner EBLNS et PGLNS

Dans cette section, nous combinons les voisinages de EBLNS et PGLNS dans une nouvelle approche, et évaluons son efficacité.

5.3.1 Guidé par la propagation et basé sur les explications.

Ce candidat est une combinaison de (1) `exp-obj`, (2) `exp-cft`, (3) `pgn`, (4) `repgn` et (5) `rapgn`. Nous agrégeons simplement les voisinages composants les deux approches précédemment testées. `rpgn` est préféré à `ran`, comme voisinage aléatoire, à cause de sa

robustesse [85]. Chaque voisinage est appliqué séquentiellement jusqu'à ce qu'une nouvelle solution soit trouvée. Ce candidat est nommé *PaEGLNS*.

Nous avons également évalué une version adaptative de PaEGLNS, qui applique un voisinage en fonction de son aptitude à construire de nouvelles solutions. Cependant, cette approche ne s'est pas révélée compétitive avec l'approche séquentielle, et nous n'avons pas reporté les résultats la concernant.

Le tableau 5.6 montre les résultats de l'évaluation de PaEGLNS. La moyenne arithmétique de la valeur de l'objectif (obj) et l'amplitude (amp) de PaEGLNS sont indiquées, les valeurs de la précédente évaluation de PGLNS et EBLNS y sont également reportées.

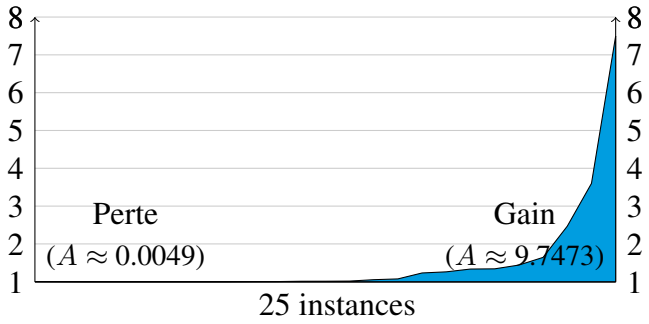
TABLE 5.6 – Évaluation de PaEGLNS en comparison avec PGLNS et EBLNS.

Instance	PaEGLNS obj	PaEGLNS amp (%)	PGLNS obj	PGLNS amp (%)	EBLNS obj	EBLNS amp (%)
Problème de minimisation						
cars_cars_4_72	*109.3*	4.57	*109.3*	5.49	118	2.54
cars_cars_16_81	119.6	9.20	121.2	2.48	**117.1**	11.10
cars_cars_26_82	127.8	4.69	**127.5**	5.49	139.5	3.58
cars_cars_41_66	*108.8*	1.84	*108.8*	1.84	110	0
cars_cars_90_05	**299.9**	11.34	300.8	11.97	321	0
fastfood_ff3	1573.4	31.78	3883.4	90.64	**1330**	0
fastfood_ff58	**1185**	9.28	8882.7	101.78	1373	0
fastfood_ff59	**242**	0	871	0	319.2	21.30
fastfood_ff61	**153.9**	12.35	221.7	24.81	355	0
fastfood_ff63	**110**	34.55	146.9	40.16	331	0
league_model20-3-5	*49984*	0	*49984*	0	*49984*	0
league_model30-4-6	*79973*	0	*79973*	0	85974	23.26
league_model50-4-4	*99971*	0	*99971*	0	118971.7	33.62
league_model55-3-12	**110949.2**	9.01	139949	35.73	136950.5	29.21
league_model90-18-20	1169916.1	8.55	1922916	7.28	**1152924.8**	26.89
league_model100-21-12	3086911.2	4.54	3139911.9	0	**2651923.1**	80.70
rcpsp_11	**80**	3.75	81	3.70	81.8	1.22
rcpsp_12	38.5	2.60	**38.3**	2.61	39.3	2.54
rcpsp_13	*78*	0	*78*	0	*78*	0
rcpsp_14	*140.9*	0.71	*140.9*	0.71	141	0
vrp_A-n38-k5.vrp	**1901.1**	22.36	2341.6	22.16	2114	36.94
vrp_A-n62-k8.vrp	5879.8	15.36	6318.1	6.66	**5840.4**	27.87
vrp_B-n51-k7.vrp	4072.4	30.13	4291.8	15.87	**3950.8**	30.42
vrp_P-n20-k2.vrp	**299.9**	38.68	402.1	41.28	364.2	17.57
vrp_P-n60-k15.vrp	**2262.7**	13.66	2271.9	12.28	2316.3	24.87
Problème de maximisation						
mario_mario_easy_2	*628*	0	*628*	0	*628*	0
mario_mario_easy_4	**507.6**	15.96	506	8.70	502.7	3.38
mario_mario_n_medium_2	**806.9**	42.76	719.9	31.95	610.5	125.31
mario_mario_n_medium_4	693.9	31.56	576.6	52.90	**711.9**	30.90
mario_mario_t_hard_1	*4783*	0	*4783*	0	1426.1	282.87
pattern_set_mining_k1_ionosphere	**36.6**	87.43	16.9	130.18	29.2	89.04
pattern_set_mining_k2_audiology	**53.8**	1.86	53.6	1.87	38.2	2.62
pattern_set_mining_k2_german-credit	6.4	93.75	3	0	**7.1**	169.01
pattern_set_mining_k2_segment	22.2	72.07	11.8	8.47	**28**	0
pc_25-5-5-9	**64.2**	1.56	62.8	3.18	64.1	3.12
pc_28-4-7-4	**54.5**	23.85	47.7	12.58	49	46.94
pc_30-5-6-7	**60.8**	13.16	60	0	58.7	13.63
pc_32-4-8-0	**109.8**	20.04	104.7	14.33	108.8	10.11
pc_32-4-8-2	88.7	23.68	84.9	15.31	**91.4**	6.56
ship-schedule.cp_5Ships	**483650**	0	483645.5	0.01	483239	0.24
ship-schedule.cp_6ShipsMixed	**300664**	1.47	300185	4.86	253749	41.92
ship-schedule.cp_7ShipsMixed	**399832**	3.15	396137	20.40	254441.5	102.45
ship-schedule.cp_7ShipsMixedUnconst	**385896**	3.11	364141.5	21.01	269851	41.39
ship-schedule.cp_8ShipsUnconst	**833724**	0.19	782516	20.51	539091	35.92
still-life_09	*42*	0	37.6	13.30	*42*	0
still-life_10	**53.3**	3.75	49.8	4.02	53.2	3.76
still-life_11	61.3	4.89	59	0	**61.7**	1.62
still-life_12	74.4	5.38	63.3	4.74	**74.6**	5.36
still-life_13	85.9	4.66	79.7	2.51	**86.4**	6.94

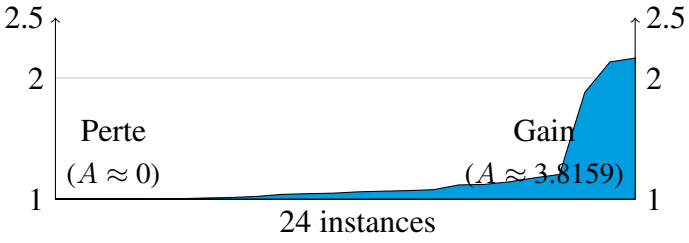
PaEGLNS trouve des solutions équivalentes, ou meilleures, dans 69.4 % des instances traités (34 sur 49). Sur les 16 autres instances, PaEGLNS est toujours classé second, presque toujours derrière EBLNS. De plus, PaEGLNS est plus performant que PGLNS dans 77.5 % des instances traitées, et fait mieux que EBLNS dans 65.3 % des instances traitées. En général, combiner les voisinages de PGLNS et EBLNS est profitable en terme de qualité des solutions, mais également en terme de stabilité. Dans 58.8 % des instances mieux résolues avec PaEGLNS, l'amplitude est inférieure à 5 %, ce qui signifie que quasiment toutes les résolutions aboutissent à la même dernière solution.

Nous allons maintenant mesurer les gains à choisir PaEGLNS plutôt que PGLNS ou EBLNS. La figure 5.5 reporte les coefficients multiplicateur entre PGLNS et PaEGLNS.

FIGURE 5.5 – Coefficient multiplicateur entre PGLNS et PaEGLNS, par instance.



(a) Problèmes de minimisation.

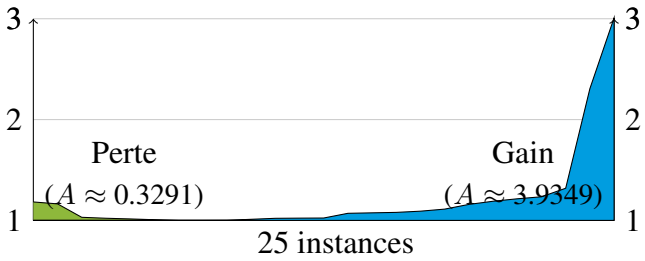


(b) Problèmes de maximisation.

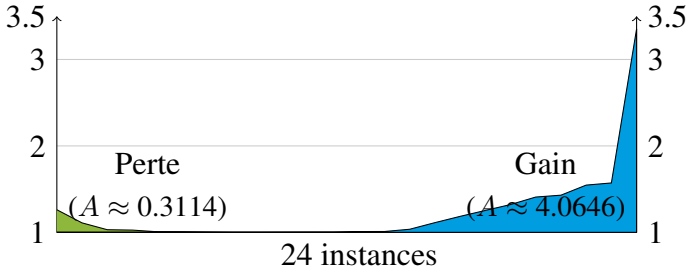
Quelle que soit la typologie du problème traité, l'apport de PaEGLNS par rapport à PGLNS est sans appel : il s'agit clairement de l'approche la plus intéressante, tout problèmes confondus. On ne note presqu'aucune dégradation des solutions, seulement celles liées à `rcpsp_12` et `cars_cars_26_82` et qui se traduit par une région verte de surface non nulle ($A \approx 0.0049$) dans le graphique 5.5(a). La comparaison avec EBLNS (figure 5.6) est légèrement moins avantageuse, bien que toujours largement en faveur de PaEGLNS. Il ne fait pas de doute que choisir PaEGLNS est gage d'efficacité.

Sans véritable surprise, les voisinages génériques (guidés par la propagation et basés sur les explications) sont complémentaires : d'une part, ceux guidés par la propagation construisent le graphe des dépendances entre variables et détectent, dans le problème traité, les sous-parties étroitement liées. D'autre part, ceux basés sur les explications aident à se concentrer sur les solutions partielles "facile à réparer" et "facile à améliorer" en révélant les relations existantes entre la variables objectif et les variables de décisions. Combiner ces voisinages permet de bénéficier des avantages des deux approches et apporte plus de stabilité dans l'obtention des solutions.

FIGURE 5.6 – Coefficient multiplicateur entre EBLNS et PaEGLNS, par instance.



(a) Problèmes de minimisation.



(b) Problèmes de maximisation.

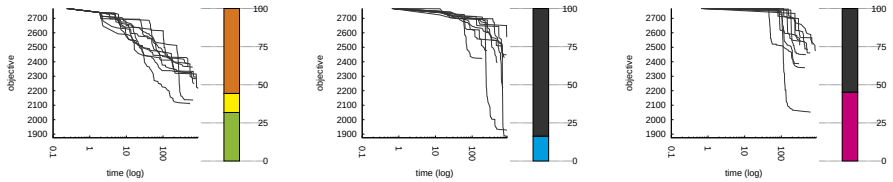
5.4 Analyse en profondeur

Dans cette section, nous mettons en valeur certains résultats qui corroborent les précédents résultats trouvés jusqu'alors. Chaque figure présentée dans cette section est composée d'un graphique et d'un histogramme. Le graphique illustre l'évolution de la valeur de la variable objectif tout au long de la résolution (dans une échelle logarithmique) des dix exécutions de la même approche (PGLNS, cftLNS, objLNS, EBLNS et PaEGLNS). L'histogramme reporte la répartition moyenne des voisinages du candidat utilisés pour résoudre une instance donnée. Le code couleur est le suivant :

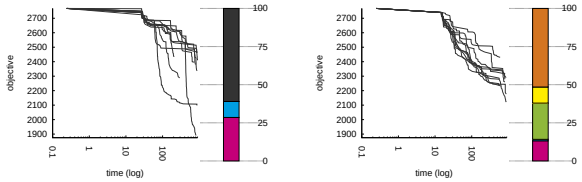
- `exp-obj` est en rose,
- `exp-cft` est en bleu,
- `ran` est en gris,
- `pgn` est en vert,
- `repgn` est en jaune et,
- `rapgn` est en orange.

Concernant le problème Vehicle Routing (Figure 5.7), les histogrammes indiquent qu'une grande proportion des solutions intermédiaires trouvées est due au voisinage aléatoire. Sa proportion varie 54 % et 83 %. Concernant PaEGLNS, qui trouve la meilleure solution, on observe le même phénomène : même si sa contribution est réduite, le voisinage aléatoire permet de trouver environ la moitié des nouvelles solutions (Figure 5.7 (e)). Le voisinage aléatoire apporte une forte diversification mais ne permet pas à lui seul d'avancer dans la résolution de ce problème. Les deux autres voisinages les plus utiles sont `exp-obj` et `pgn`.

FIGURE 5.7 – Résolution de l'instance `vrp_P-n60-k15.vrp` avec les cinq approches.

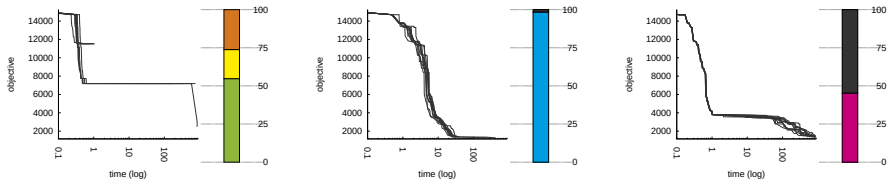


(a) PGLNS (b) cftLNS (c) objLNS

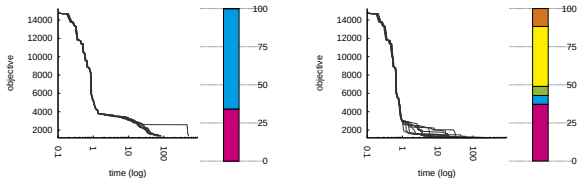


(d) EBLNS (e) PaEGLNS

FIGURE 5.8 – Résolution de l'instance `fastfood_ff58` avec les cinq approches.



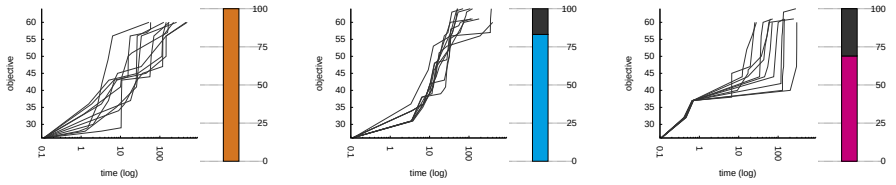
(a) PGLNS (b) cftLNS (c) objLNS



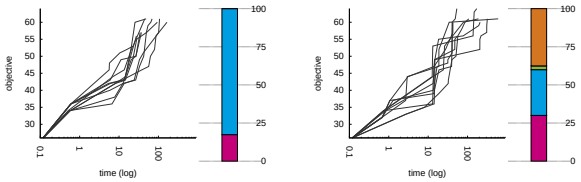
(d) EBLNS (e) PaEGLNS

Concernant le problème Restaurant Assignment Problem (Figure 5.8), où EBLNS fonctionne bien, la contribution de `ran` à la progression de la recherche est négligeable ($< 1\%$) en comparaison avec celles de `exp-obj` (34.2 %) et `exp-cft` (65.7 %). La stabilité de EBLNS sur cette instance est clairement visible sur le graphique (Figure 5.8 (b)). On observe sensiblement le même phénomène pour cftLNS. Par contre, objLNS repose à plus de 50 % sur le voisinage aléatoire. Là où EBLNS peine, PaEGLNS améliore non seulement l'objectif, par rapport aux autres approches, mais il traite l'instance plus rapidement (cf. tableau 5.5 et tableau 5.6). Étonnement, `exp-obj` et `rpgn`, qui n'ont pas l'air d'être clé dans les autres approches, contribuent à hauteur de deux tiers à l'obtention de nouvelles solutions, et la contribution de `exp-cft` est significativement réduite dans PaEGLNS en comparaison avec EBLNS (Figure 5.8 (e)). Une fois encore, il semble que plus il y a de voisinages combinés, meilleures sont les solutions.

FIGURE 5.9 – Résolution de l'instance `pc_30-5-6-7` avec les cinq approches.



(a) PGLNS (b) cftLNS (c) objLNS



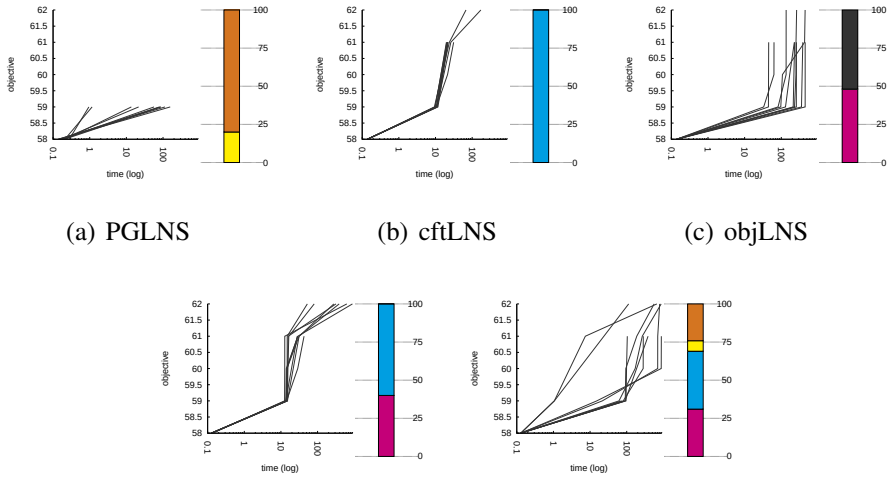
(d) EBLNS (e) PaEGLNS

Sur le problème Prize Collecting (Figure 5.9), nous observons que la résolution repose seulement sur `rapgn` pour PGLNS (Figure 5.9 (a)). Concernant les autres combinaisons, au contraire, la contribution du voisinage purement aléatoire à la découverte de solution est assez faible. Un autre point remarquable est la proportion de `exp-obj` et `exp-cft` dans EBLNS et PaEGLNS : plus de 60 % des solutions trouvées par ces deux candidats reposent sur les voisinages expliqués. Cependant, au vu des résultats de PaEGLNS, il est clair qu'il est pertinent de combiner des voisinages variés.

Enfin, concernant le problème Still Life Problem (Figure 5.10), utiliser `exp-obj` et `exp-cft` est la clé du succès (Figure 5.10 (b), Figure 5.10 (c) et Figure 5.10 (d)). Seulement, une dégradation des performances est observée lorsque d'autres voisinages sont utilisés en complément (Figure 5.10 (e)). Cela montre qu'il peut exister un risque à combiner des voisinages variés, cela peut nuire à la sémantique de chacun d'entre eux. Par contre, cette observation est rarement faite dans nos expérimentations, et ne permet pas d'infirmer le bénéfice des combinaisons.

Toutes les figures relatives aux autres instances traitées sont fournies en annexe. Il est possible de voir que, sur les problèmes Restaurant Assignment et Itemset Mining et, dans une moindre mesure, sur les problèmes Price Collecting et Still Life, EBLNS est assez stable. De plus, excepté sur le problème Vehicle Routing, la contribution du voisinage aléatoire `ran` à

FIGURE 5.10 – Résolution de l'instance `still-life_11` avec les cinq approches.



(a) PGLNS (b) cftLNS (c) objLNS

(d) EBLNS (e) PaEGLNS

la progression de la recherche pour EBLNS est marginale.

Dans cette section, nous avons évalué `exp-obj` et `exp-cft`, deux voisinages pour LNS basés sur les explications. Nous avons limité les expérimentations aux modèles sans contraintes globales, et aucune des approches évaluées ici n'a bénéficié des possibles améliorations en terme de filtrage ni d'explications. Nous avons montré que objLNS, cftLNS et EBLNS sont compétitives avec PGLNS sur une grande plage de problèmes d'optimisation. Même si ces combinaisons à base de voisinages expliqués sont globalement moins stables et légèrement plus lents, elles permettent de résoudre des problèmes de PGLNS traite mal. En plus, nous avons confronté ces premiers résultats avec PaEGLNS, et avons montré que combiner plusieurs voisinages variés permet d'améliorer les résultats, apportant plus de stabilité. Une telle conclusion confirme les études précédemment effectuées : une grande diversification contribue au fonctionnement global de LNS.

6

Conclusion et travaux futurs

Dans cette partie, notre motivation principale était de proposer des voisinages pour LNS qui soient indépendants de l'instance traitée et qui ne nécessitent pas de configuration préalable à leurs utilisations. Pour cela, nous avons choisi d'exploiter les explications de deux manières différentes. D'une part, en forçant le conflit entre le chemin de décisions menant à une solution et la coupe qui en découle, elles servent à construire des voisins plus facilement extensibles à de nouvelles solutions. D'autre part, les explications servent à concevoir des instanciations partielles plus à même d'être améliorées en expliquant pourquoi la variable objectif n'a pas pu être affectée à une meilleure valeur dans la solution courante. C'est là une nouvelle manière d'exploiter les explications. Nous avons également exposé comment un système d'explications asynchrone et paresseux pouvait être mis en place afin d'en alléger l'utilisation. Ensuite, nous avons évalué plusieurs combinaisons de voisinages sur un ensemble de dix problèmes d'optimisation (et 39 instances) variés. Ces expérimentations ont montré que les voisinages que nous proposons sont non seulement compétitifs avec les voisinages guidés par la propagation, références en terme de voisinages génériques, mais également qu'il était tout à fait pertinent de les combiner tous ensemble. En particulier, PaE-GLNS a tout à fait sa place lorsqu'il s'agit de choisir une heuristique de voisinage pour la recherche à voisinage large qui soit générique et efficace. Ces résultats sont encourageants et doivent être étendus à un plus large panel de problèmes. Cette étude a fait l'objet d'une présentation au programme doctoral de la conférence CP [94] et d'une publication dans le journal Constraints [95].

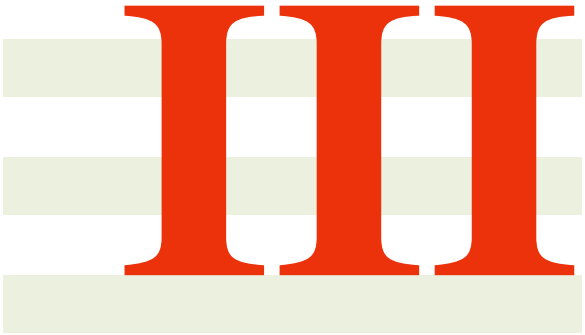
Néanmoins, les évaluations que nous avons menées ne mesurent pas l'influence du modèle sous-jacent et de ses particularités sur les performances de LNS, d'autant plus avec les voisinages basés sur les explications. La résolution des instances du problème Modified Car Sequencing avec PGLNS donne des résultats assez déroutants où la part d'aléatoire est manifeste dans l'obtention de nouvelles solutions. On peut imaginer que l'absence de contraintes globales nuit à ces voisinages, tout comme elle nuit certainement aux voisinages à base d'explications. Nous nous sommes interdits la modélisation de problèmes à l'aide de contraintes globales, dont les bénéfices seraient pourtant nombreux : de meilleurs algorithmes de filtrage, des explications plus précises, moins de variables intermédiaires produisant des voisins plus compacts, etc. Il est tout à fait vraisemblable que la stratégie de branchement influe sur les voisinages génériques étudiés dans cette partie. Ces suppositions doivent être validées dans

des travaux futurs. Il est également envisageable de compléter la stratégie de *fast restarts*, communément employée, avec une analyse de la qualité des voisins produits par les heuristiques, quel que soit leur type (génériques, à base d'explications, adhoc, etc.). Si l'échec levé lors de l'extension d'un voisin est corrélé à la variable objectif, il peut être intéressant d'évaluer dans quelle mesure ceci est dû à la partie libre du voisin, auquel cas l'échec est imputable à la stratégie de branchement. Au contraire, si l'échec est attribuable à la partie figée du voisin, il est plus efficace d'en chercher un autre plus *sain*, au lieu d'attendre d'avoir atteint la limite de redémarrage. Cette approche peut, toutefois, souffrir de trop consulter la base d'explications. En parallèle, il parait également intéressant d'étudier l'influence du moteur de propagation sur la qualité des explications. Il est évident que la façon dont les événements sont relayés au travers du réseau de contraintes domine la manière dont les explications sont calculées, puisque celle-ci repose sur l'ordre dans lequel les domaines sont modifiés. Cependant, l'efficacité de la recherche à voisinage large repose en partie sur le nombre de voisins qu'il est possible de tester dans une unité de temps. Plus ce nombre sera élevé, et plus grande sera sa chance de réussite. Dans une telle situation, alourdir les voisinages ou trop reposer sur les solveurs de contraintes pour trouver une solution n'est pas forcément gage de performance. Enfin, il pourrait également être intéressant d'implémenter et d'évaluer nos voisinages dans un solveur à génération paresseuse de clauses.

Dans le cadre de nos expérimentations, nous avons testé d'alimenter une base de nogoods au fur et à mesure de la progression de la recherche à voisinage large pour éviter de découvrir, à de trop nombreuses reprises, les sous-arbres dont on sait qu'ils ne résultent en aucune solution. Cette technique s'est avérée rarement compétitive avec les autres approches présentées dans ce chapitre, et ceci pour de simples raisons d'efficacité. En effet, la propagation devenait de plus en plus lourde à propager à mesure que la base de nogoods[1] grossissait. D'autre part, nous avons également expérimenté une autre approche qui exploite d'avantage le solveur de contraintes. Après plusieurs essais infructueux à chercher une solution à partir de multiples voisins, la limite imposée par le *fast restart* était levée, et l'exploration de l'espace de recherche était laissé en mode recherche arborescente, reposant entièrement sur la stratégie de branchement déclarée. Les expérimentations préliminaires ont donné des résultats intéressants dont la plus-value était majoritairement de rendre la résolution complète.

En conclusion, l'une des contributions fondamentales de ces travaux réside dans l'implémentation d'un solveur de contraintes nativement expliqué, Choco-3.1.0 [91, 90]. À la différence de PaLM [52], qui était une extension d'une précédente version de Choco-1, cette version du solveur est nativement expliquée, en proposant un système d'explications interne asynchrone et débranchable. L'implémentation de ce système d'explications s'est faite à l'initiative de Narendra Jussien et Laurent Perron.

[1] Nous nous sommes limités, pour l'évaluation, aux nogoods générés sur les redémarrages, tels qu'ils sont décrits par Lecoutre et autres [67].



Un Moteur de Propagation Configurable

7

Introduction

> « *DSLs can serve a big role for the future of optimization, and constraint programming in particular, as they can encode in a simple format the knowledge that is nowadays confined in [...] obscure parts of solvers implementations.* »
>
> (Pierre Schaus & Jean-Noël Monette, 2013, COSpeL – CP'13)

Dans l'introduction de sa présentation lors de l'ACP Research Excellence Award (CP'13), Jean-Charles Régin a largement insisté sur les axes à considérer pour bien résoudre un problème. Il affirme que le gain en performance se joue sur trois aspects, dont les chances de réussite sont inversement proportionnelles à l'espérance de gain[1]. Il place en première position la modélisation, comme étant le levier avec la plus grande espérance de gain, mais dont les chances de succès sont les plus faibles. En pratique, l'utilisateur dispose, pour modéliser, d'un choix entre plusieurs types de variables et de manières de les représenter, et surtout d'une librairie de contraintes, dont chacune est caractérisée par sa sémantique, sa puissance de filtrage et son efficacité algorithmique. Vient ensuite le parcours de l'espace de recherche, qui n'est, bien entendu, pas à négliger, bien que les espérances de gains soient moins importantes. En pratique, l'utilisateur peut décrire facilement les stratégies de branchements et d'exploration de l'espace de recherche, à l'aide de *goals* [49] ou de *search combinators* [102]. En dernier lieu vient le travail sur les structures de données. En aucun cas, il n'évoque la manière dont les contraintes doivent être propagées, mécanisme pourtant central des solveurs de contraintes, finalement peu considéré dès lors qu'on cherche à mieux résoudre un problème. Cela s'explique assez simplement : comme la résolution d'un problème est dominée par la modélisation, d'une part, et la stratégie de recherche, d'autre part, la marge de gain restante est comparativement assez faible –sauf en ce qui concerne les cas pathologiques. De plus, et surtout, un solveur de contraintes est rarement instrumenté pour permettre, ou simplement faciliter, la description du moteur de propagation. D'ailleurs, comparativement aux autres éléments constitutifs d'un solveur de contraintes, le moteur de propagation est globalement peu étudié et méconnu. Parmi les raisons, on peut citer, par exemple, la difficulté de son évaluation (les stress-tests sont rarement représentatifs d'un

[1] "Simple solutions for complex problems", Jean-Charles Régin. ACP Research Excellence Award, CP2013 (Uppsala, Suède).

fonctionnement en conditions réelles, les choix d'implémentation des algorithmes de propagation ont un impact sur les performances, etc.), l'hétérogénéité des contraintes décrivant un modèle qui force à déporter certaines améliorations au sein des propagateurs eux-mêmes, ou encore les freins conceptuels, essentiellement liés aux différences entre solveurs (l'orientation du moteur en tête).

L'algorithme définissant la propagation des contraintes a été décliné en plusieurs classes [8] ; il s'agit principalement de versions modernisées d'algorithmes de cohérence d'arc [69, 75, 48]. De tels algorithmes sont caractérisés par leurs *orientations* et par l'ordre de révisions qu'ils spécifient. L'orientation détermine l'information nécessaire à la propagation des réductions de domaines au travers du réseau de contraintes, *p. ex.*, les variables dont le domaine a été modifié ou bien les contraintes susceptibles de filtrer. Les algorithmes de propagation les plus communément implantés sont soit *orientés variables*, soit *orientés contraintes* (ou *propagateurs*). Par exemple, IBM CPLEX CP Optimizer [49], Choco [62, 91], Minion [37] et or-tools [82] sont basés sur un algorithme orienté variable, alors que Gecode [109], SICStus Prolog [24] et JaCoP [60] sont basés sur un algorithme orienté propagateur. Puisqu'ils orchestrent la propagation, un moteur de propagation se doit d'être très efficace, ses caractéristiques impactent généralement le solveur dans sa totalité. Il doit, entre autres, être capable de supporter un nombre élevé d'opérations à la seconde, ce qui implique que les marges d'améliorations structurelles, a priori faibles, peuvent être déterminantes. La moindre amélioration, ou dégradation, se répercute sur toutes les résolutions. Pour choisir le prochain propagateur à exécuter, il faut prendre en compte deux informations souvent orthogonales : la puissance de filtrage et la rapidité d'exécution. Autrement formulé, il convient de trouver le plus court chemin entre deux points fixes, non pas en terme de distance mais de temps, sans connaître *a priori* les valuations de chaque arête et en ne s'interdisant pas d'emprunter plusieurs fois la même arête ou le même sommet. Le développement d'un moteur de propagation, et plus généralement d'un solveur de contraintes, requiert de faire des choix et des compromis pour conjuguer adaptabilité (résoudre des problèmes divers) avec efficience (les résoudre efficacement). Il ne faut pas omettre que la conception du moteur de propagation dépend de la conception des propagateurs, et vice versa. Dès lors, le modifier peut rapidement devenir une tâche complexe. En effet, implémenter un algorithme de propagation qui soit à la fois correct, efficace, en adéquation avec l'interface spécifique du solveur et son langage de programmation relève parfois du défi. Vraisemblablement, seuls les développeurs du solveur sont à mêmes de le faire évoluer tout en maintenant ces garanties. En conséquence, les moteurs de propagation tendent à devenir figés, et inaccessibles à l'étape de modélisation. Il est assez rare d'adapter le moteur de propagation aux particularités des problèmes traités[2]. Cependant, rendre le moteur de propagation aussi flexible et ouvert que peut l'être la stratégie de branchement et d'exploration, doit permettre de le démystifier, de le vulgariser auprès des utilisateurs, tout en augmentant la fiabilité des outils.

La description des moteurs de propagation à l'aide d'un langage a pour double ambition de faciliter leurs conceptions, dans une optique de prototypage, et de lister leurs garanties et propriétés. Bien entendu, il n'est pas nécessairement question de donner un accès aux algorithmes de propagation aux utilisateurs débutants. Cependant, un tel outil peut rapidement devenir pertinent pour un développeur ou un modeleur avancé, en s'inscrivant dans une démarche "*Write Once Read Many*". Le premier pourra enrichir le solveur, en travaillant une fois pour toutes sur les structures de données critiques, dans un périmètre restreint, et en les mettant à disposition de tous. Le second voudra rapidement tester et valider une stratégie de propagation en limitant le développement et les bogues, sans se préoccuper de la manière dont le moteur sera effectivement mis en pratique. De plus, chacun aura la possibilité d'en

[2] On notera, entre autres, les travaux sur la propagation de contraintes arithmétiques [72].

bénéficier pleinement et simplement dans d'autres solveurs.

Dans ce chapitre, nous proposons un langage indépendant de tout solveur et permettant de configurer un moteur de propagation à l'étape de modélisation. Notre contribution est triple. Tout d'abord, nous présentons un langage dédié à la description de moteurs de propagation de contraintes. Nous montrons qu'il permet d'exprimer un large éventail de stratégies existantes. Les propriétés de notre langage dédié sont exploitées pour assurer la correction des moteurs produits. Puis, nous présentons une implantation concrète de notre langage, basée sur Choco et une extension du langage MiniZinc [36]. Enfin, nous montrons que le surcoût induit par notre langage et son implantation est opérationnellement acceptable pour prototyper des moteurs de propagation.

Le Chapitre 8 est consacré à la description du langage dédié. Après avoir présenté succinctement ce qu'est un Langage Dédié (Section 8.1), nous définissons l'ensemble des techniques et services que le solveur cible doit fournir pour supporter entièrement le langage (Section 8.2). Puis, nous présentons le langage dédié à la description des moteurs de propagations (Section 8.3). La Section 8.4 est dédiée aux garanties des moteurs de propagation produits. Enfin, nous discutons de son implémentation au sein du langage de modélisation MiniZinc, et des possibles limites de notre approche (Chapitre 8.5.1). Le Chapitre 9 est dédié à l'évaluation du langage. Après avoir décrit les moteurs dont l'évaluation est comparée à leur version *native* (Section 9.1), nous évaluons la phase d'interprétation des descriptions ainsi que leurs utilisations pour résoudre des problèmes extraits de la distribution MiniZinc (Section 9.2). Enfin, nous montrons, combien, à l'aide du langage dédié à la propagation, il est facile et rapide de décrire un comportement adapté au traitement d'un problème particulier, celui de la règle de Golomb (Section 9.3).

8

Un langage dédié à la description des moteurs de propagation

Sommaire

Dans l'optique d'exprimer, ou décrire, convenablement un moteur de propagation, nous introduisons un *Langage Dédié* [113]. Premièrement, nous définissons la notion de langage dédié. Puis, nous établissons la liste des prérequis pour bénéficier de ce langage. Ensuite, nous présentons notre langage dédié en deux étapes : (1) la déclaration des groupes d'arcs et (2) la structuration et combinaison de ces groupes. Nous listons les garanties et propriétés de notre langage dédié, et proposons une manière de l'implémenter. Enfin, après avoir présenté MiniZinc [36], nous décrivons comment l'étendre pour supporter notre langage.

8.1 Qu'est-ce qu'un langage dédié ?

En informatique, un langage dédié, ou DSL (pour "*Domain Specific Language*" en anglais) est un langage de programmation dont les spécifications sont dédiées à la résolution de problèmes propres à un domaine d'application précis, et uniquement ce domaine. Il devient pertinent de définir un langage dédié, et le logiciel le supportant, dès lors que l'utilisation d'un langage simplifie la description d'un problème (par rapport à un langage déjà existant, si un tel langage existe) et que la problématique apparait suffisamment régulièrement. Les langages dédiés ont donc un potentiel intéressant en terme de productivité et de fiabilité [112]. Dans [117], les auteurs établissent qu'un DSL doit être *universel*, c'est-à-dire qu'il

FIGURE 8.1 – Extrait tiré du langage dédié décrit dans la Section 8.3.
⟨*group_decl*⟩ : := ⟨*id*⟩ **:** ⟨*predicates*⟩ **;**

doit permettre de modéliser et résoudre tous les problèmes du domaine auquel il est dédié. Il doit également avoir une logique naturelle pour résoudre les problèmes.

La description des *règles syntaxiques* d'un langage de programmation peut se faire sous la forme de Backus-Naur (abrégée en BNF). Un telle notation distingue les méta-symboles (*c.-à-d.* les symboles de BNF), les terminaux et les non-terminaux. Les symboles non-terminaux sont les noms des catégories que l'on définit, tandis que les terminaux sont des symboles du langage décrit. Ces symboles, ainsi que les règles de production (paires formées d'un non-terminal et d'une suite de terminaux et de non-terminaux) constituent la grammaire formelle du langage.

Dans la Figure 8.1, nous présentons la règle syntaxique définissant la structure d'un groupe, extraite du DSL décrit en détail dans la Section 8.3. ⟨*group_decl*⟩, ⟨*id*⟩ et ⟨*predicates*⟩ sont non-terminaux. **:** et **;** sont des terminaux. ::= est un méta-symbole signifiant "est défini par". D'autres méta-symboles existent, tels que |, (ou), et facilitent la description des règles syntaxiques. La liste des méta-symboles utilisés pour décrire notre langage est donnée dans la Section 8.3.

8.2 Spécificités techniques requises

Afin de pouvoir implémenter le DSL (décrit dans la Section 8.3) dans un solveur de contraintes, certaines fonctionnalités doivent être supportées par le solveur cible. Parce qu'elles rendent les solveurs globalement plus efficaces, ces caractéristiques sont vraisemblablement déjà présentes dans les solveurs modernes.

- Les propagateurs à priorités [106] : les propagateurs sont discriminés par rapport à leur priorité. La priorité doit permettre de déterminer quel est le prochain propagateur à exécuter : les propagateurs les plus "légers" (au sens de la complexité) sont exécutés avant les plus "lourds", l'ordre peut être raffiné en fonction de la puissance de filtrage ;

- Groupes de propagateurs [64] : une politique de propagation spécifique est définie pour un ensemble de propagateurs, ce dernier est connu du moteur de propagation grâce à un propagateur *pilote* qui le référence ;

- Un accès ouvert aux propriétés des variables et propagateurs : par exemple la cardinalité d'une variable, l'arité d'un propagateur ou sa priorité.

Notons par ailleurs que l'évaluation des propriétés décrites dans ce dernier point doit pouvoir être faite dynamiquement, pendant la propagation. Bien entendu, une évaluation dynamique induit un coût et peut requérir des structures de données spécifiques [19].

Pour apporter plus de flexibilité et de précision dans l'expression des moteurs, nous admettons que tous les arcs de A, l'ensemble des arcs d'un CSP, sont explicitement accessibles. Ils peuvent être représentés explicitement et des *watched literals* [38] ou des *advisors* [63] peuvent leur être associés. Une représentation explicite des tous les arcs se fait au détriment de l'occupation mémoire : $\mathcal{O}(|X| \times |P|)$ objets supplémentaires sont nécessaires, chacun d'entre eux maintenant deux pointeurs, l'un vers une variable, l'autre vers un propagateur. Cependant, rendre accessibles les arcs amène plus de flexibilité en donnant conjointement

accès aux propriétés de leur variable et de leur propagateur. De plus, les arcs peuvent être organisés par variable, par propagateur ou par propriété (par exemple, par priorité).

8.3 Un language pour décrire les moteurs de propagation

Le langage, introduit dans cette Section, s'utilise une fois la modélisation du problème terminée, avant la phase de résolution. Il repose sur les arcs du modèle à résoudre, sur lequel il définit un ordre total, et permet de décrire une politique de propagation qui servira à la création d'un moteur de propagation à utiliser par le solveur cible.

Le langage est constitué de deux sous-parties hiérarchisées (Figure 8.2). Il convient, tout d'abord, de constituer des groupes d'arcs distincts (Section 8.3.1). Puis, en combinant ces groupes entre eux, la structure globale du moteur de propagation est décrite (Section 8.3.2). Nous présenterons ces deux étapes individuellement dans la suite.

FIGURE 8.2 – Le point d'entrée du langage dédié.
⟨*propagation_engine*⟩ : := ⟨*group_decl*⟩+ ⟨*structure_decl*⟩ **;**

La syntaxe est présentée en BNF[1], avec les conventions suivantes : un terme en police typewriter `tt` indique un symbole terminal ; un terme en italique encadré de chevrons ⟨*it*⟩ indique un symbole non terminal ; des crochets [*e*] définissent une option ; des doubles crochets [[a-z]] définissent un intervalle ; le symbole plus *e*+ définit une séquence de une ou plus répétitions de *e* ; le symbole étoile *e*⋆ définit une séquence de zéro ou plus répétitions de *e* ; l'ellipse *e*,... définit une séquence non vide de *e* séparés par des virgules ; l'alternance *a*|*b* indique des alternatives.

8.3.1 La déclaration des groupes

Nous présentons maintenant le langage dédié à la déclaration des groupes. La notion de groupe est définie dans [64]. La notion de groupe a un objectif simple mais puissant : contrôler un ensemble de propagateurs en une fois en définissant, pour le groupe, une politique interne de planification et d'exécution. Ici, nous substituons aux propagateurs des arcs.

Constituer des groupes d'arcs doit se faire dans le respect des règles d'affectation (décrites dans la Figure 8.3). Bien que le DSL manipule des arcs, l'utilisateur n'a accès qu'aux variables et aux contraintes lors de la modélisation pour définir les groupes. Les variables et contraintes, et leurs propriétés, peuvent ainsi être directement référencées dans cette partie du DSL. Les propagateurs, quant à eux, ne sont habituellement pas accessibles à l'étape de modélisation et ne peuvent donc être référencés qu'au travers de leurs propriétés. Les propriétés que nous avons réunies dans cette sous-partie du DSL sont présentes et consultables dans la majorité des solveurs.

Un groupe est déclaré avec l'aide d'un identifiant et d'une liste de prédicats. L'identifiant ⟨*id*⟩ est un nom unique commençant par une lettre, par exemple `G1`. Un prédicat est une fonction booléenne $a \rightarrow \{\texttt{true}, \texttt{false}\}$, elle définit la condition d'appartenance à un groupe. Un prédicat est dit *en extension* s'il repose sur une variable ou une contrainte (via `var_id` et `cstr_id`). Les arcs associés à une variable ou une contrainte sont éligibles pour le groupe.

[1] La forme de Backus-Naur [3] est une notation permettant de décrire les règles syntaxiques des langages de programmation.

FIGURE 8.3 – Définition des groupes

⟨*group_decl*⟩ : := ⟨*id*⟩ **:** ⟨*predicates*⟩**;**
⟨*id*⟩ : := [[a-zA-Z]][[a-zA-Z0-9]]⋆
⟨*predicates*⟩ : := ⟨*predicate*⟩ | `true`
| (⟨*predicates*⟩ ((`&&` | `||`) ⟨*predicates*⟩)+)
| ! ⟨*predicates*⟩
⟨*predicate*⟩ : := `in` (⟨*var_id*⟩ | ⟨*cstr_id*⟩)
| ⟨*attribute*⟩ ⟨*op*⟩ (⟨*int_const*⟩ | `"string"`)
⟨*op*⟩ : := == | != | > | >= | < | <=
⟨*attribute*⟩ : := `var`[`.`(`name`|`card`)]
| `cstr`[`.`(`name`|`arity`)]
| `prop`[`.`(`arity`|`priority`|`prioDyn`)]
⟨*int_const*⟩ : := [+−][[0-9]][[0-9]]⋆
⟨*var_id*⟩ : := -⋆⟨*id*⟩
⟨*cstr_id*⟩ : := -⋆⟨*id*⟩

Exemple 8.1 (Prédicats en extension) *La déclaration "*`in` (x_1)*" définit un ensemble d'arcs* A_1 *tel que pour tout arc appartenant à* A_1 *soit associé à la variable* x_1, $\forall a \in A_1, X(a) = x_1$. *La déclaration "*`in` (c_1)*" définit un ensemble d'arcs* A_2 *tel que tout arc appartenant à* A_2 *soit associé à la contrainte* c_1, $\forall a \in A_2, P(a) \in P(c_1)$.

Un prédicat est dit *en intension* s'il repose sur une ou plusieurs propriétés. Tous les arcs retenus doivent alors satisfaire les propriétés. Les prédicats en intension sont construits à l'aide de trois paramètres : un attribut ⟨*attribute*⟩, un opérateur ⟨*op*⟩ et une valeur entière ou une chaîne de caractères. Un ⟨*attribute*⟩ fait référence à une propriété d'une variable (`var.name`, le nom d'une variable ; `var.card`, sa cardinalité), d'une contrainte (`cstr.name`, le nom d'une contrainte ; `cstr.arity`, sa cardinalité) ou d'un propagateur (`prop.arity`, l'arité d'un propagateur ; `prop.priority`, sa priorité statique ; `prop.prioDyn`, sa priorité dynamique[2]). Les prédicats peuvent, bien sûr, être combinés à l'aide de trois opérateurs booléens `&&`, `||` and `!`.

Exemple 8.2 (Prédicats en intension) *La déclaration "*`prop.priority` < 4*" définit un ensemble d'arcs* A_1 *tel que tout arc présent dans* A_1 *ait une priorité statique strictement inférieure à 4.*

La définition des groupes peut être dépendante du modèle, lorsque des variables ou des contraintes d'un problème sont référencées explicitement grâce à leurs noms. Mais elle peut également être totalement indépendante lorsque les groupes sont décrits grâce à des propriétés. D'ailleurs, les spécificités d'un solveur peuvent être facilement prises en compte par le DSL, en étendant ⟨*attribute*⟩. Le but de ⟨*groups_decl*⟩ est de construire des sous-ensembles d'arcs. Cette partie du DSL est évaluée de manière impérative, *c.-à-d.* chaque évaluation de prédicat peut déplacer un ou plusieurs arcs de A vers le sous-ensemble défini par le prédicat. Cependant, un arc peut répondre `true` à des prédicats de différents groupes. Pour empêcher d'avoir à discriminer les prédicats entre eux, cette partie du DSL est évaluée impérativement de haut en bas et de droite à gauche (cf. Propagation Unique 8.5). Ainsi, un arc qui répond `true` à un prédicat devient indisponible pour les prédicats suivants.

[2]Les propagateurs indiquent une priorité statique qui leur est propre, qualifiant la complexité de leur algorithme. La priorité dynamique prend en compte le nombre de variables du propagateur qui ne sont pas instanciées [106]

8.3.2 Déclaration de la structure

La seconde partie du DSL (Figure 8.4) porte sur la description, basée sur les groupes, des structures de données. Le but ici est double : indiquer "où" placer un arc lorsqu'il doit être planifié et "comment" sélectionner le prochain qui doit être exécuté. De la déclaration structurelle résulte un arbre syntactique hiérarchisé basé sur une collection ⟨*coll*⟩ qui servira à la création du moteur de propagation.

FIGURE 8.4 – Structure d'un moteur de propagation

⟨*structure*⟩ ::= ⟨*struct_ext*⟩ | ⟨*struct_int*⟩
⟨*struct_ext*⟩ ::= ⟨*coll*⟩ `of {`⟨*elt*⟩`, ...}` [`key` ⟨*comb_attr*⟩]
⟨*struct_int*⟩ ::= ⟨*id*⟩ `as` ⟨*coll*⟩ `of {`⟨*many*⟩`}` [`key` ⟨*comb_attr*⟩]
⟨*elt*⟩ ::= ⟨*structure*⟩
| ⟨*id*⟩ [`key` ⟨*attribute*⟩]
⟨*many*⟩ ::= `each` ⟨*attribute*⟩ `as` ⟨*coll*⟩ [`of {`⟨*many*⟩`}`]
[`key` ⟨*comb_attr*⟩]
⟨*coll*⟩ ::= `queue(`⟨*qiter*⟩`)`
| [`rev`] `list(`⟨*liter*⟩`)`
| [`max`] `heap(`⟨*qiter*⟩`)`
⟨*qiter*⟩ ::= `one` | `wone`
⟨*liter*⟩ ::= ⟨*qiter*⟩ | `for` | `wfor`
⟨*comb_attr*⟩ ::= (⟨*attr_op*⟩ `.`)⋆⟨*ext_attr*⟩
⟨*ext_attr*⟩ ::= ⟨*attribute*⟩ | `size`
⟨*attr_op*⟩ ::= `any` | `min` | `max` | `sum`

Collections

Une ⟨*coll*⟩ est définie par une structure de données abstraite (SDA) et elle est composée d'autres SDA et d'arcs. Il est alors envisageable qu'une ⟨*coll*⟩ contienne des arcs mais aussi d'autres ⟨*coll*⟩. Une SDA définit où les éléments vont être planifiés et comment ils sont sélectionnés pour une exécution. Il en existe trois types : une `queue`, une `list` et un `heap`. Une file (`queue`) est une collection d'éléments de la forme *first-in-first-out*. Une liste (`list`) est une collection d'éléments ordonnés statiquement (`rev` renverse la liste). Un tas (`heap`) est une collection d'éléments ordonnés dynamiquement grâce à un critère (la fonction de comparaison par défaut est $\leq$, le mot-clé `max` le transforme en $\geq$). Structurellement, chaque type de SDA indique où placer un élément nouvellement à planifier. Il faut également définir la manière dont les ⟨*coll*⟩ vont être *parcourus*.

Définition 8.1 (Parcours) *Un* parcours *décrit l'ordre dans lequel les éléments planifiés dans une* ⟨*coll*⟩ *sont sélectionnés, dé-planifiés et renvoyés pour être parcourus à leurs tours.*

Parcourir un arc revient simplement à exécuter son propagateur suite à la modification de sa variable. Les parcours sont décrits en associant un itérateur à une ⟨*coll*⟩.

Les itérateurs de bases `one` et `wone` sont définis pour toute ⟨*coll*⟩. L'itérateur `one` parcours un seul élément d'une ⟨*coll*⟩, en respectant l'ordre structurel induit par le SDA.

Exemple 8.3 *L'instruction "`queue(one)`" sélectionne, dé-planifie et renvoie le plus vieil élément planifié pour qu'il soit parcouru à son tour.*

L'itérateur `wone`, version courte pour 'while one', appelle `one` jusqu'à ce qu'il n'y ait plus d'éléments planifiés dans la ⟨*coll*⟩. Une liste définit deux itérateurs supplémentaires : `for` and `wfor`. L'itérateur ⟨*for*⟩ parcourt séquentiellement les éléments d'une liste dans l'ordre croissant de son index, comparable à un *balayage*. L'itérateur `for` n'autorise pas de retours arrières ni d'interruption de la traversée. Les ⟨*coll*⟩ sont mutables ; de nouveaux éléments peuvent être planifiés dans une ⟨*coll*⟩ pendant l'itération. Comme l'itérateur ⟨*for*⟩ ne permet pas de revenir en arrière, il n'y a pas de garantie que la ⟨*coll*⟩ est vide à la fin de l'itération. C'est pourquoi nous introduisons ⟨*wfor*⟩, version courte pour 'while for', qui appelle `for` jusqu'à ce que la liste soit vide.

Exemple 8.4 *Étant donnée une liste dans laquelle deux arcs a_1 et a_2 sont planifiés et ordonnés lexicographiquement, un parcours de cette liste avec l'instruction ⟨for⟩ équivaut à exécuter, dans l'ordre, a_1 puis a_2. Considérons que l'exécution de a_2 planifie a_1 et un troisième arc a_3, alors a_1 sera ignoré par l'itérateur ⟨for⟩ (l'indice de a_1 dans la liste est inférieur à celui de a_2 en cours de parcours), a_3 sera exécuté (l'indice de a_3 dans la liste est supérieur à celui de a_2 en cours de parcours). Au contraire, l'itérateur ⟨wfor⟩, une fois a_3 exécuté, vérifiera que la liste n'est pas vide auquel cas, le liste sera de nouveau parcourue.*

Un tas (`heap`) requiert la déclaration d'un critère de comparaison. Le critère de comparaison est optionnel lorsqu'il s'agit d'une collection (`list`), par défaut les éléments ne sont pas triés et l'ordre de déclaration reste inchangé. Dans le cas d'une liste, les éléments sont triés une fois la liste remplie, c'est pourquoi il convient de déclarer un critère de comparaison statique. Dans le cas d'un tas, l'ordre défini par le critère de comparaison est maintenu dynamiquement à chaque nouvel ajout d'un élément dans le tas, c'est pourquoi il convient de déclarer un critère de comparaison dynamique. Comme il n'y a aucune garantie qu'une ⟨*coll*⟩ contienne exclusivement des arcs, le critère de tri doit être défini par :

- la combinaison d'un mot-clé (`key`) et d'un attribut d'arc (⟨*attribute*⟩) ou
- la combinaison d'un mot-clé et d'attributs combinés (⟨*comb_attr*⟩).

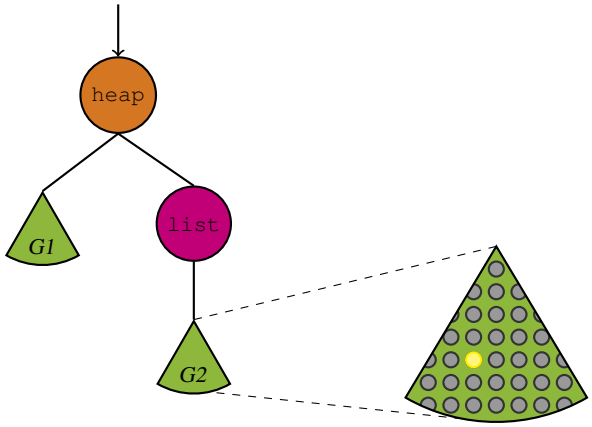
Les attributs combinés permettent de valuer un groupe, soit grâce à une fonction (⟨*min*⟩, ⟨*max*⟩ ou ⟨*sum*⟩), soit en définissant un élément représentatif et évaluable du groupe (⟨*any*⟩). L'évaluation d'un ensemble est faite en évaluant les arcs de l'ensemble (grâce à ⟨*attr_op*⟩). Une extension de ⟨*attribute*⟩, nommée ⟨*ext_attr*⟩, est introduite dans la Figure 8.4. Une ⟨*ext_attr*⟩ permet d'atteindre tout attribut d'un élément d'une collection d'élément, comme sa taille `size`.

Exemple 8.5 *La Figure 8.5 déclare un moteur de propagation basé sur un tas, la Figure 8.6 représente visuellement ce moteur sous la forme d'un arbre. Le tas haut niveau impose que les éléments qui le composent soit évaluables (comparables entre eux). Tous les arcs de `G1` sont évalués en retournant la cardinalité de leur variable (Ligne 2). Le second groupe est une liste d'arcs `G2` (Ligne 3-5). Une liste n'est pas évaluable en tant que telle, son évaluation doit alors être déléguée à l'un des arcs qui la compose. Donc, un arc de `G2` est sélectionné arbitrairement (`any`) pour obtenir la cardinalité de sa variable (Ligne 5). Le mot-clé `any` indique à la fois de déléguer l'évaluation du critère à un élément de `list` et de le choisir arbitrairement. Visuellement, cela correspond aux éléments situés en bas à droite de la figure : le cercle vert est constitué d'arcs du groupe `G2` (les cercles gris) dont l'un d'entre eux (le cercle jaune) a été sélectionné arbitrairement pour représenté le groupe . Au sein de la liste, les arcs sont triés par ordre lexicographique du nom de leur variable (Ligne 4).*

FIGURE 8.5 – Déclaration d'un moteur de propagation basé sur un tas.

```
1. heap(wone) of {
2.     G1 key var.card,
3.     list(for) of {
4.           G2 key var.name
5.     } key any.var.card
6. }
```

FIGURE 8.6 – Représentation sous la forme d'un arbre du moteur décrit dans la Figure 8.5.



Structures

Nous allons maintenant décrire comment les structures de données abstraites sont hiérarchisées pour décrire le moteur de propagation. Le point d'entrée d'une déclaration de structure est ⟨*structure*⟩, elle permet de décrire la structure du plus haut niveau qui pilotera la propagation. Une ⟨*structure*⟩ peut être définie soit en extension (⟨*struct_ext*⟩) ou en intension (⟨*struct_int*⟩).

Une ⟨*struct_ext*⟩ définit une collection ⟨*coll*⟩ composée d'une liste d'éléments ⟨*elt*⟩. Un ⟨*elt*⟩ peut faire référence soit à l'identifiant d'un groupe et une instruction d'ordre (⟨*id*⟩ [`key` ⟨*attribute*⟩]), soit à une autre⟨*structure*⟩.

Exemple 8.6 (⟨*struct_ext*⟩) *La Figure 8.5 déclare un moteur de propagation basé sur un tas, la Figure 8.6 représente visuellement ce moteur sous la forme d'un arbre. La structure de haut niveau, le tas, est décrite en extension en listant les éléments qui la compose : les arcs du groupe d'arcs* `G1` *et une liste. Cette dernière est également décrite en extension en listant ses éléments, c.-à-d. les arcs du groupe* `G2`*.*

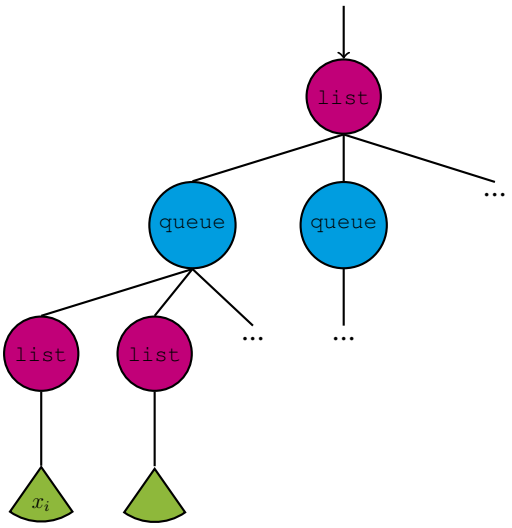
Une ⟨*struct_int*⟩ définit une structure régulière basée sur un quantifieur unique `each-as`[3]. Une ⟨*struct_int*⟩ permet d'exprimer, de manière compacte, des structures imbriquées, là où l'utilisation de ⟨*struct_ext*⟩ impliquerait une expression verbeuse et sujette à l'introduction d'erreurs. ⟨*struct_int*⟩ indique que chacun des arcs d'un groupe doit être affecté, sous certaines conditions données par un attribut, à une collection. Elle prend en entrée un ensemble d'arcs défini par un identifiant de groupe et elle *génère* autant de ⟨*coll*⟩ que requis par l'instruction ⟨*many*⟩, potentiellement imbriquées les uns dans les autres. L'instruction ⟨*many*⟩ implique une itération sur les valeurs de l'⟨*attribute*⟩ et affecte à la même ⟨*coll*⟩ les éléments avec la même valeur pour l'⟨*attribute*⟩.

[3]Ce quantifieur trouve son origine dans l'instruction GROUP BY du langage SQL.

FIGURE 8.7 – Déclaration d'une structure régulière.

```
1. G as list(wfor) of{
2.   each prop.priority as queue(for) of{
3.     each var as list(for)}
4. }
```

FIGURE 8.8 – Représentation sous la forme d'un arbre de la structure régulière déclarée dans la Figure 8.7.



Exemple 8.7 (⟨*struct_int*⟩) *La Figure 8.7 montre la déclaration d'une structure régulière, et la Figure 8.8 sa représentation visuelle sous la forme d'un arbre. Les arcs du groupe* `G` *vont servir au déploiement de la structure de données. Il va y avoir autant de sous-ensembles (disjoints) d'arcs de* `G` *qu'il y a de priorités différentes (“*`prop.priority`*”) parmi les propagateurs présents dans* `G` *(Ligne 2). Pour rappel, Schulte et Stuckey en définissent 7 [106]. Puis, chaque sous-ensemble est traité un par un et les arcs qu'il contient sont de nouveau regroupés par variables (mot-clé “*`var`*”) dans des listes (Ligne 3). Les listes impliquant des arcs associés à des propagateurs de même priorité sont ajoutées dans une même file (“*`queue(for)`*”, Ligne 2) correspondant à une* `priority`*. Les files sont ajoutées dans la liste de haut niveau qui impose un parcours particulier (“*`list(wfor)`*”, Ligne 1) : la file de plus petite priorité doit être parcourue avant celle de priorité suivante. Si un critère dynamique le nécessitait, l'étape d'affectation à la bonne collection pourrait être effectuée pendant la propagation.*

8.4 Propriétés et garanties du langage

Un langage dédié apporte des propriétés, mais également des garanties. Nous décrivons maintenant toutes les propriétés et garanties de notre langage.

Propriété 8.1 (Description indépendante du solveur) *La description des moteurs de propagation est indépendante du solveur cible.*

Un langage dédié bénéficie naturellement d'un haut niveau d'abstraction. Le fait qu'il ne repose sur l'architecture d'aucun solveur (exception faite des prérequis listés dans la Section 8.2) permet de l'implanter facilement dans de nombreux solveurs de contraintes.

Lorsqu'une partie seulement de prérequis est supportée par le solveur cible, le langage dédié est moins expressif.

- Accès aux propriétés : tous les solveurs modernes donnent un accès non restreint aux propriétés des variables (*p. ex.*, la taille de son domaine) et des propagateurs (*p. ex.*, son arité), ces éléments sont cruciaux dès lors qu'il s'agit de gérer finement la propagation ;
- Les propagateurs à priorités : l'absence de qualification de la complexité algorithmique des propagateurs interdit de définir des moteurs où le choix du prochain propagateurs à executer tient compte de cette propriétés. On peut raisonnablement estimer que fournir une telle information nécessite peu de développement ;
- Groupe d'arcs : le solveur cible donne accès aux arcs du problème mais ne supporte pas la notion de groupe, c'est-à-dire qu'il interdit les structures imbriquées. Dans ce cas, l'utilisation du langage est réduite à la définition de l'ordre de révision, et ceux pour tous les arcs du problème, sans hiérarchisation possible ;
- Représenter explicitement les arcs : le solveur cible ne propose pas de représenter explicitement tous les arcs d'un problème. Dans ce cas, le DSL perd en expressivité, une partie du DSL concernant la description des groupes devient inutilisable, et les moteurs produits sont moins précis. De plus, les orientations déclarables sont limitées à celles supportées par le solveur cible, *c.-à-d.* la majeure partie du temps, une seule. L'utilisateur peut toujours déclarer des groupes grossiers, et hiérarchiser ces groupes entre eux.

Les solveurs dit "orientés variables" sont vraisemblablement mieux préparés pour la représentation des arcs, puisque les propagateurs ont certainement été conçus pour réagir précisément à la modification d'une variable donnée. Les solveurs dit "orientés propagateurs", par contre, ont été conçus de façon à ce que les propagateurs réagissent globalement, ils peuvent cependant calculer les variables modifiées depuis leurs derniers réveils, grâce aux *advisors* par exemple. Dans ce cas, une refonte de cœur du solveur et des propagateurs est à envisager pour remplir les conditions préliminaires à l'implantation du langage. Il est évident qu'une telle refonte est critique pour les développeurs du solveur. Cependant, les solveurs gérant les événements fins sont plus à même de supporter ce genre de modification majeure. De plus, une telle opération apporte non seulement plus de flexibilité dans le solveur mais ouvre également de nouvelles perspectives dans le processus de résolution. Naturellement, cela ne remet pas en cause les moteurs *natifs* (*c.-à-d.* , qui ne reposent pas sur le langage), au contraire, cela vient en complément de ceux-ci.

Propriété 8.2 (Expressivité) *Le langage dédié à la déclaration des moteurs de propagation est expressif.*

Les structures de données abstraites les plus communément utilisées sont présentes dans le DSL. Elles permettent de décrire des moteurs simples, comme une simple file de variables, mais également structurellement complexes, comme la liste de files à priorité. De plus, l'accès aux propriétés du solveur rend possible la conception des fonctionnements aboutis, mélangeant plusieurs orientations par exemple, ou en décrivant des orientations difficilement représentables à ce jour (*p. ex.*, orienté arité). Une grande variété de moteurs de propagation peut alors être décrite de manière concise.

Propriété 8.3 (Extensibilité) *Le langage dédié à la déclaration des moteurs de propagation peut être étendu.*

Nous proposons, dans ce DSL, les propriétés que tous solveurs modernes devraient mettre à disposition des utilisateurs. Il est toutefois envisageable de le faire évoluer pour s'adapter aux spécificités d'un solveur. Le langage peut être étendu de deux manières. Premièrement, il est possible de définir de nouveaux attributs pour rendre la définition des groupes plus concise, ou, tout simplement, possible. Deuxièmement, de nouvelles structures de données et de nouveaux itérateurs peuvent être définis pour décrire de nouveaux schémas de propagation.

Propriété 8.4 (Garantie de couverture) *Tous les arcs d'un modèle sont représentés dans le moteur de propagation produit.*

L'expressivité de notre langage autorise la déclaration de moteurs de propagation incomplets, dans lequel un ou plusieurs arcs peuvent être absents. Dans ce cas, une partie du modèle sera inconnue du moteur de propagation produit, et donc, ignorée lors de la propagation. Ce défaut de couverture est détectable lors de l'interprétation de la description, en maintenant simplement le compteur des arcs affectés à un groupe, et celui des groupes effectivement utilisés. L'utilisateur est alors averti d'un défaut de couverture et la construction du moteur est interrompue. Une alternative est de compléter automatiquement la déclaration d'un moteur de propagation par un groupe au prédicat universel collectant les arcs orphelins et de lui attribuer un comportement par défaut.

Propriété 8.5 (Propagation unique) *Un arc ne peut être planifié qu'une seule fois par événement.*

Lors de l'évaluation de l'expression d'un moteur de propagation, un arc peut satisfaire plusieurs prédicats et donc pourraient être affectés à plusieurs groupes. L'évaluation haut-bas gauche-droite du langage garantit qu'un arc n'est affecté qu'à un seul groupe, le premier dont il satisfait le prédicat. Ainsi, un arc ne peut être planifié qu'une seule fois par événement. Si un arc pouvait être planifié plusieurs fois, cela pourrait, d'une part, détériorer les performances du moteur, et d'autre part, et surtout, désynchroniser les propagateurs incrémentaux. Dès lors qu'un arc est dupliqué, l'information qu'il contient devient redondante et non pertinente. Par exemple, propager une seconde fois l'information que la borne de la variable Y passe de 9 à 8 sur la contrainte $X < Y$ ne sert à rien. L'occupation mémoire que sa duplication implique et l'inutilité de cette duplication nuisent aux performances du moteur. D'autre part, la notion d'incrémentalité des contraintes reposent sur la condition qu'un même événement n'est reçu qu'une fois. Si cette condition n'est pas respectée, les structures internes d'une contrainte peuvent être désynchronisées. Par exemple, si une contrainte doit maintenir incrémentalement le nombre de ses variables qui sont instanciées, et qu'au cours d'une propagation elle reçoit plusieurs fois le même information d'instantiation pour une unique variable, le compteur devient faux. En ce sens, l'implantation du langage dans un solveur ne modifie en rien la complexité standard de la phase de propagation.

Propriété 8.6 (Conformité) *Les arcs associés à une paire propagateur-variable inexistante dans le modèle ne sont pas représentables dans le moteur de propagation.*

Grâce à cette propriété, aucun arc sans signification, au sens du modèle, ne peut ainsi être planifié et donc propagé pendant la phase de propagation. Cette propriété ne détériore pas la complexité temporelle du moteur de propagation, puisqu'elle est vérifiée à l'interprétation d'une expression.

Propriété 8.7 (Complétude) *La complétude est assurée par un itérateur complet, tel que* `wone` *et* `wfor`*, associé à la collection top-niveau.*

Ainsi, quelques soient les itérateurs associés aux collections *filles* composant le moteur de propagation, la collection *mère* garantit qu'il ne reste aucun élément planifié à la fin de son parcours, *c.-à-d.* quand la phase de propagation est terminée, et que la phase de recherche va être appliquée. Sélectionner un itérateur tel que `one` or `for` peut supprimer cette garantie de complétude. Cependant, Katriel, dans [55], explique précisément en quoi une propagation incomplète peut être bénéfique. D'autre part, Boussemart et autres [21] ont étudié comment la propagation pouvait être arrêtée avant d'atteindre son point fixe, avec un contrôle statistique.

Propriété 8.8 (Complexité) *Les deux propriétés de "Conformité" et de "Propagation unique" assurent que la complexité de l'algorithme de propagation sous-jacent n'est pas dégradée.*

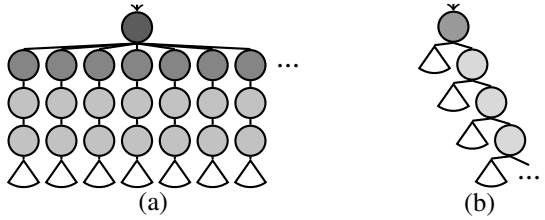
Cependant, la déclaration de moteurs de propagation mal formés ou l'usage excessif de critères à évaluer dynamiquement peut avoir un effet non-négligeable sur les performances globales du moteur de propagation. Les utilisateurs doivent être conscients que la généralisation de l'architecture des moteurs de propagation peut nuire à l'efficacité pendant la phase de prototypage et surtout pendant la phase d'évaluation. Pour autant, le langage empêche de créer des moteurs dont la propagation ne se terminent pas.

Propriété 8.9 (Fiabilité) *Le langage dédié décrit des moteurs de propagation fiables.*

Parce qu'il respecte toutes les propriétés listées jusqu'alors, le DSL produits des moteurs fidèles aux descriptions de l'utilisateur. Cependant, il donne aussi la possibilité de décrire des moteurs de propagation mal formés, bien que syntactiquement corrects. Par exemple, l'un des groupes peut être vide, un groupe peut être défini mais pas utilisé dans le moteur, plusieurs structures imbriquées peuvent être basées sur le même attribut, un groupe peut être composé d'un seul arc, etc.

Exemple 8.8 (Moteurs de propagation mal formés) *Dans la Figure 8.9 (a), les collections constituées d'un seul élément peuvent être fusionnées avec la collection parente. Dans la Figure 8.9 (b), suivant les itérateurs utilisés, la structure peut être mise à plat. Par exemple, deux* `lists` *avec le même itérateur* `wone` *peuvent être agrégées.*

FIGURE 8.9 – Exemples de moteurs de propagation mal formés.



Dans la plupart des cas, un contrôle peut être fait de manière à informer l'utilisateur de telles malformations. Il est envisageable d'appliquer automatiquement, à l'interprétation d'une expression, les simplifications correspondantes, si elles existent : supprimer les groupes vides, compacter les structures imbriquées inutiles, agréger les groupes composés d'un seul arc, etc.

FIGURE 8.10 – Le modèle MiniZinc du problème de la `séquence magique`, étendu avec notre langage.

```
1 : include "globals.mzn";
2 : annotation engine(string: s);
3 : annotation name(string: s);
4 : int: n;
5 : array [0..n - 1] of var 0..n: x;
6 : constraint count(x,0,x[0])::name("c0");
7 : constraint forall (i in 1..n - 1) (count(x, i, x[i]));
8 : solve
9 : :: engine("
10 : G1: in(\"c0\");
11 : All: true;
12 : list(wone) of {queue(wone) of{G1},
13 :      All as queue(wone) of {each cstr as list(wfor)}};
14 : ")
15 : satisfy;
```

8.5 Implémentation

Dans cette section, nous présentons l'implémentation de notre langage au dessus d'un solveur cible. Étendre un langage de modélisation est une manière simple et naturelle d'implanter notre langage : il donne accès aux objets du modèle, telles que les variables et les contraintes, tout en maintenant l'indépendance vis-à-vis du solveur cible.

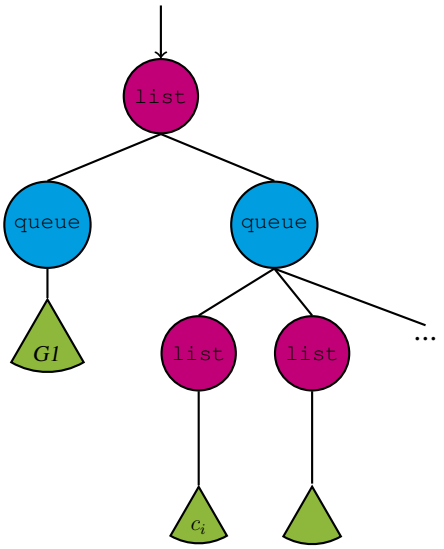
ANTLR pour interpréter le langage Nous avons sélectionné ANTLR [81] pour interpréter la grammaire définie par notre langage. ANTLR, pour "*ANother Tool for Language Recognition*", est un framework libre de construction de compilateurs utilisant une analyse LL(*). ANTLR sépare les trois étapes d'interprétation ; l'analyseur syntaxique (le *lexeur*), la reconnaissance des structures de phrases (le *parseur*) et la traduction en instructions du solveur (l'*arbre syntaxique abstrait*). Une telle séparation permet de n'avoir à adapter que l'arbre syntaxique abstrait au solveur cible, le lexeur et le parseur n'ayant pas à être modifiés.

MiniZinc pour implanter le langage Nous avons choisi d'étendre le langage de modélisation MiniZinc [36]. Il s'agit d'un langage de modélisation de problèmes de satisfaction de contraintes simple mais expressif. Il est supporté par un nombre important de solveurs modernes.

En pratique, il est nécessaire d'ajouter la description du moteur de propagation directement dans le modèle MiniZinc. Cela se fait à l'aide de deux annotations créées à cet effet. La première sert à référencer les contraintes du modèle (Figure 8.10, Ligne 2). La seconde sert à déclarer le moteur de propagation (Figure 8.10, Ligne 3). Même si MiniZinc permet la définition d'annotations plus complexes, telles que celles utilisées pour déclarer les stratégies de recherche, les annotations présentées ici sont basiques, basées sur des *strings*. Ainsi, on préserve l'indépendance de notre langage vis-à-vis du langage de modélisation.

Exemple 8.9 (Séquence magique) *La figure 8.10 montre un exemple de modèle MiniZinc dans lequel la déclaration d'un moteur de propagation a été ajoutée, et la Figure 8.11 sa représentation visuelle sous la forme d'un arbre. Les lignes 1, 4-8 et 15 décrivent un modèle classique pour le problème de la séquence magique de taille 5 (prob019, [41]). La*

FIGURE 8.11 – Représentation sous la forme d'un arbre de la structure régulière déclarée dans la Figure 8.10.



déclaration des annotations nécessaires au référencement des contraintes (ligne 2) et à la déclaration du moteur de propagation (ligne 3) doit être placée dans l'en-tête du fichier. La première contrainte (ligne 6) est annotée avec `::name("c0")`*, il sera, par la suite, possible d'y faire référence. Les lignes 9-14 décrivent le fonctionnement du moteur de propagation. Deux groupes sont définis. Le premier groupe* `G1` *(ligne 10) est composé des arcs impliqués dans la contrainte nommée* `"c0"`*. Le second groupe* `All` *(ligne 11) est composé des arcs restants, grâce à l'évaluation haut-bas gauche-droite. Puis, la structure du moteur de propagation est décrite. La liste haut niveau est composée de deux files (lignes 12-13). La première file contient les arcs inclus dans le groupe* `G1` *(ligne 13), alors que la seconde file est composée de listes d'arcs organisées par contrainte (ligne 13). Il en résulte un moteur de propagation qui propage toujours en premier les arcs associés à la contrainte* `"c0"`*, avant de traiter les autres arcs du modèle.*

8.5.1 FlatZinc, cas de la décomposition de contraintes

La distribution de MiniZinc inclut un langage bas niveau destiné à être interprété par les solveurs, nommé FlatZinc. FlatZinc est le langage cible dans lequel sont traduits les modèles MiniZinc. Lors de la traduction, les annotations sont automatiquement transférées au modèle FlatZinc, incluant donc les annotations précédemment introduites. Toutefois, la traduction de MiniZinc vers FlatZinc peut induire une décomposition des contraintes définies basée sur les contraintes primitives de FlatZinc. Par exemple, en l'absence de la contrainte `AllDifferent` dans un solveur, la contrainte sera reformulée en une clique de contraintes d'inégalités. Dans l'optique de se concentrer d'abord sur l'utilisation du langage, seules les contraintes primitives de FlatZinc ainsi que les contraintes globales surchargées, c'est-à-dire supportées par le solveur cible, peuvent être utilisées. Les contraintes globales dont la définition repose sur des prédicats sont donc exclues (voir la documentation de MiniZinc pour plus de détails sur la décomposition de contraintes à l'aide de prédicats). En effet, même si la déclaration originelle pouvait être transformée de la même manière que les contraintes décomposées le sont, rien ne garantit que la déclaration résultante réponde aux intentions initiales. Considérons une description demandant de propager aussi tard que possible une contrainte `AllDifferent` qui, à cause de la réformation, est transformée en une clique de contraintes

d'inégalités : cela pourrait être complètement trompeur et contre-productif. La difficulté de reformuler des expressions déclarées avec notre langage réside essentiellement dans l'utilisation des prédicats et des attributs liés à la définition des groupes. L'utilisateur peut déclarer un moteur de propagation basé sur les priorités. Lors de la description du moteur, l'utilisateur base son raisonnement sur une contrainte globale de priorité basse. Si la contrainte n'est pas supportée par le solveur cible, elle sera reformulée à partir de contraintes primitives d'arités faibles et de priorités certainement différentes. Une alternative à l'interdiction des décompositions est d'intégrer notre langage directement dans MiniZinc, puis d'instrumenter chacune des contraintes reformulées avec la déclaration d'un groupe et d'une structure de données reposant sur ce groupe et les contraintes primitives. Ainsi, toute contrainte reformulée pourrait être nommée et incluse dans un prototype de moteur. Il est à noter également que l'impact de la décomposition sur l'efficacité de la propagation n'a pas encore été adressé formellement, et représente en soi un intéressant projet de recherche.

8.5.2 Sémantique opérationnelle

Les groupes définissent des partitions imbriquées d'arcs, et chaque politique de propagation est implantée par un itérateur spécifique. Ainsi, à la sémantique opérationnelle de n'importe quel moteur de propagation défini à l'aide de notre langage correspond à un arbre de partitions et leurs itérateurs associés. Notre implantation est alors basée sur cet arbre. Sa mise en œuvre peut être décrite par une sémantique opérationnelle. L'algorithme 8 présente la sémantique opérationnelle de la déclaration du moteur de propagation décrit dans la Figure 8.10, lignes 9-14. Il montre comment peuvent être interprétées les étapes principales de l'analyse grammaticale du langage, et les étapes principales de la résolution au sein d'un moteur interprété. Le programme maître, non détaillé ici, appelle les deux premières procédures DÉCLARATION DES GROUPES et DÉCLARATION DE LA STRUCTURE pendant l'analyse grammaticale, les deux dernières procédures (PLANIFIER and PROPAGER) sont appelées pendant la résolution, par l'algorithme de propagation. Tout d'abord, la déclaration des groupes (lignes 10) consiste en une itération sur les arcs du modèle pour affecter chacun d'entre eux à un groupe. Le prédicat d'affectation au groupe $G1$ est évalué en premier puisqu'il a été déclaré en premier. Ensuite, la déclaration de la structure (lignes 11-22) consiste en une séquence d'instructions de création et d'initialisation, des structures bas niveau ($q1$ et ls) jusqu'à la structure haut niveau ($l1$). Notons que les arcs du groupe $G1$ ont été organisés par contrainte dans une structure intermédiaire (ligne 17). L'algorithme 8 présente également une interprétation des deux fonctions principales d'un moteur de propagation : comment planifier un arc (lignes 23-24) et comment sélectionner le prochain arc à exécuter (lignes 35-50). Les structures imbriquées impliquent que la planification d'un arc déclenche aussitôt la planification de tous les groupes reliant cet arc à la collection de haut niveau. Lorsqu'un arc est choisi pour être exécuté, l'ordre implicite ou explicite entre les groupes décrit lors de la déclaration est préservé, il en va de même pour les itérateurs. Par exemple, les arcs de $G1$ sont à exécuter avant tous les autres arcs (lignes 37-40). Quand $q1$ est vide, alors $q2$ est vidé à son tour. L'itérateur `wfor` déclaré dans la Figure 8.10 ligne 13, est transformé en une combinaison d'une boucle *tant-que* et d'une boucle *pour* (lines 43-47).

Algorithme 8 Schéma d'évaluation du moteur de propagation décrit dans la Figure 8.10

Require: Arcs : l'ensemble des arcs du modèle
1: **procedure** DÉCLARATION DES GROUPES ▷ Figure 8.10, lignes 10-11
2: Declare $G1$, All comme étant des ensembles d'arcs
3: **for** chaque arc a dans Arcs **do**
4: **if** a est associé à un propagateur de c0 **then** ▷ Prédicat de $G1$
5: déplacer a dans $G1$
6: **else if** `true` **then** ▷ Prédicat de All
7: déplacer a dans All
8: **end if**
9: **end for**
10: **end procedure**

Require: $G1$, All : ensembles d'arcs
11: **procedure** DÉCLARATION DE LA STRUCTURE ▷ Figure 8.10, lignes 12-14
12: Déclarer $q1$ comme une file
13: Remplir $q1$ avec les arcs de $G1$
14: Déclarer $q2$ comme une file
15: Déclarer ls comme un tableau de listes
16: Déclarer m comme une map
17: Joindre les arcs de All aux contraintes dans m
18: Remplir ls avec les ensembles d'arcs définis par m
19: Remplir $q2$ avec les listes de ls
20: Déclarer $l1$ comme une liste
21: Remplir $l1$ avec les files $q1, q2$
22: **end procedure**

Require: a : un arc
23: **procedure** PLANIFIER ▷ Opérations à executer lors de la planification d'un arc
24: Déclarer tmp comme un objet planifiable
25: $tmp \leftarrow a$
26: **repeat**
27: **if** tmp n'est pas déjà planifié **then**
28: Ajouter tmp dans son groupe ▷ La sélection du groupe peut être dynamique
29: $tmp \leftarrow$ groupe de tmp
30: **else**
31: **return** ▷ tmp est déjà planifié, les groupes de plus haut niveau le sont également
32: **end if**
33: **until** tmp est le groupe de haut niveau ▷ Le groupe de haut niveau appartient à lui même
34: **end procedure**

35: **procedure** PROPAGER ▷ Opérations à exécuter lors d'un appel à la propagation
36: **while** $l1$ n'est pas vide **do**
37: **while** $q1$ n'est pas vide **do**
38: Retirer le premier arc a de $q1$ et l'exécuter
39: **end while**
40: **while** $q2$ n'est pas vide **do**
41: Retirer la liste l de $q2$
42: **while** l nest pas vide **do** ▷ boucle tant-que et ...
43: **for** i dans $1 \ldots$ TAILLE(l) **do** ▷ ... boucle pour afin de mimer le comportement "wfor"
44: Retirer et exécuter le i$^{\text{ème}}$ arc a de l, si présent
45: **end for**
46: **end while**
47: **end while**
48: **end while**
49: **end procedure**

9

Expérimentations

Dans cette section, nous évaluons la phase d'interprétation du DSL sur un ensemble de problèmes extraits de la distribution MiniZinc [36]. Nous comparons également, sur la même base de problèmes, les performances de moteurs *natifs* avec leurs versions interprétées. Bien entendu, les moteurs interprétés ne peuvent pas être plus efficaces que leurs modèles natifs. Cependant, on prévoit un surcoût induit faible et limité dû, d'une part, à la phase d'interprétation de l'expression du moteur et, d'autre part, aux indirections et évaluations de critères liées à l'instanciation de l'expression dans le solveur. En effet, la plupart du temps, les structures de données utilisées dans un moteur de propagation ont des complexités très faibles. Par exemple, une file ou un bitset définit les opérations `add()` et `remove()` en temps constant, un tas binaire définit les opérations `add()`, `extractMin()` et `update()` en $O(log(n))$, où n est la taille du tas. Même si l'efficacité de ces opérations est critique, notre thèse est que l'efficacité d'un moteur de propagation réside tout autant dans l'ordre dans lequel les propagateurs sont exécutés.

La configuration Toutes les expérimentations ont été faites sur un Macbook Pro avec un 6-core Intel Xeon cadencé à 2.93Ghz sur MacOS 10.6.8, et Java 1.6.0_35. Les résultats sont basés sur 20 exécutions (l'écart-type est toujours inférieur à 5%) et les tableaux reportent le temps nécessaire à l'analyse grammaticale du fichier MiniZinc (en millisecondes) et le temps de résolution en secondes.

Le Solveur L'évaluation de notre DSL et son interprétation ont été implantées dans le solveur de contraintes Choco-3.1.0 [90, 91]. Les sources ont été modifiées pour répondre aux prérequis listés dans la Section 8.2. De plus, trois moteurs de propagation natifs (*c.-à-d.* non basés sur le langage) ont été implantés dans la bibliothèque : un moteur orienté propagateurs `prop`, un autre orienté variables `var`, le dernier est un moteur orienté propagateurs, composé de sept files et évaluant la priorité des propagateurs dynamiquement `7qd` [106].

Les expérimentations La première évaluation a été faite sur la base de 39 instances extraites de 21 problèmes de la distribution MiniZinc [36] : `bacp`, `bibd`, `black-hole`, `CostasArrays`, `debruijn-binary`, `fastfood`, `fillomino`, `ghoulomb`, `golomb`, `jobshop`, `langford`, `latin-squares`, `magicseq`, `market_split`, `nonogram`,

`p1f`, `pentominoes`, `radiation`, `rcpsp`, `slow_convergence`, `still_life`. Leurs caractéristiques sont indiquées dans le Tableau 9.1. Nous indiquons le nombre de variables créées, le nombre de propagateurs déclarés, le nombre d'arcs qu'il en résulte. Nous donnons également le nombre de solutions trouvées et le nombre de failures. Nous rappelons que tous les moteurs de propagation comparés ici construisent exactement le même arbre de recherche : les mêmes points fixes sont atteints bien que les ordres de propagation soient différents d'un moteur à l'autre. Ensuite, nous présenterons un cas d'utilisation, basé sur le

TABLE 9.1 – Liste des problèmes traités.

Problème	# variables	# propagators	# arcs	# failures	# solutions
bacp-14	1961	2027	6376	263558	1
bacp-15	1961	2020	6362	64273	1
bibd_15_03_01	12600	19580	43050	4472	1
black-hole_0	104	89	379	150174	1
black-hole_12	104	89	379	16274	1
CostasArray_14	210	367	839	10751	1
CostasArray_15	240	423	973	105885	1
debruijn_binary_02_08	4611	4356	11269	0	1
debruijn_binary_02_09	10243	9732	25093	0	1
fastfood_ff56	568	536	1281	88692	162
fastfood_ff75	652	633	1490	132349	109
fillomino_08	872	1206	3478	18974	1
fillomino_12	1835	2400	7155	82044	1
ghoulomb_3-7-16	223642	128639	511128	7342	4
golomb_09	45	47	163	8295	10
golomb_10	55	57	201	48882	10
jobshop_jobshop_la03	501	1175	2350	102929	1787
jobshop_jobshop_la18	1001	2350	4700	248071	4339
langford_l_2_09	360	659	1350	6426	0
langford_l_2_10	440	812	1660	35661	0
latin-squares-lp_12	1872	576	7056	549	1
latin-squares-lp_15	3600	900	13725	572	1
magicseq_030	1830	1830	4530	1599	1
magicseq_040	3240	3240	8040	1470	1
market_split_s4-04	30	4	120	982641	1
market_split_s4-10	30	4	120	3106859	1
non_non_fast_1	2500	100	5000	2724	1
non_non_fast_8	3025	110	6050	8508	1
p1f_10	902	965	4384	6926	27
pentominoes-int_05	66	78	858	2205	1
pentominoes-int_06	65	77	845	89	1
radiation_04	781	569	1847	72759	1
radiation_08	878	635	2077	271685	1
rcpsp_max_psp_j10_33	3867	6127	15952	52690	9
slow_convergence_0800	1602	321201	642401	0	1
slow_convergence_0900	1802	406351	812701	0	1
slow_convergence_1000	2002	501501	1003001	0	1
still_life_3x8	436	612	1586	16947	3
still_life_5x5	426	597	1553	9884	6

problème de la `Règle de Golomb`. Il permettra de démontrer combien le langage simplifie le prototypage de moteur de propagation.

9.1 Descriptions des moteurs

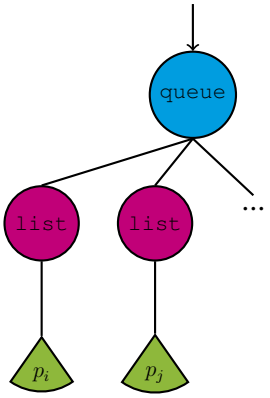
Cette section est dédiée à la description des moteurs natifs (`prop`, `var` et `7qd`) à l'aide du langage, ainsi qu'à la comparaison du traitement des fichiers MiniZinc avec les moteurs natifs et avec les moteurs interprétés. Pour chaque moteur natif, sa version interprétée construit le même arbre de recherche et préserve l'ordre ainsi que le nombre d'exécutions des propagateurs. Cela nous permet d'illustrer l'expressivité du langage.

Tout d'abord, nous présentons comment déclarer `prop`, un moteur de propagation orienté propagateurs, à l'aide du langage (Figure 9.1 et Figure 9.2). La structure prend tous les arcs du problème en entrée. Puis, les arcs sont regroupés par propagateur dans des listes. Chaque liste définit une traversée incomplète (`for`) mais la file haut niveau assure la complétude de la propagation (`wone`). Ensuite, nous présentons la déclaration de `var`, un moteur de pro-

FIGURE 9.1 – Déclaration d'un moteur de propagation orienté propagateur.

```
1 : All :true;
2 : All as queue(wone) of {
3 :     each prop as list(for)
4 : }
```

FIGURE 9.2 – Représentation sous la forme d'un arbre du moteur déclaré dans la Figure 9.1.

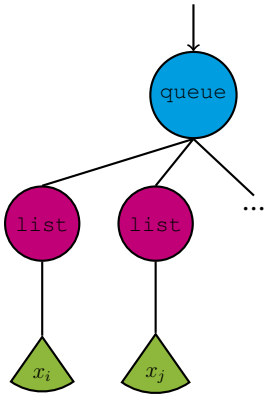


pagation orienté variables, à l'aide du langage (Figure 9.3 et Figure 9.4). Il est très proche de la description précédente, seulement, les arcs sont regroupés par variable. Enfin, nous présentons comment déclarer `7qd` à l'aide de notre langage, un moteur orienté propagateur, composé de sept files et évaluant la priorité des propagateurs dynamiquement (Figure 9.5 et Figure 9.6). Comme les précédents, il prend tous les arcs du problème en entrée. Ces arcs sont répartis dans les différentes files en fonction de la priorité dynamique de leurs propagateurs. Pour chaque priorité disponible, les arcs sont regroupés autour des propagateurs dans les listes. Comme l'évaluation des priorités est dynamique, le choix de la file dans laquelle mettre une liste est fait grâce à l'évaluation du propagateur d'un arc de la liste. La liste de haut niveau garantit à la fois la complétude de la propagation (`wone`) et que les événements associés aux propagateurs de faible priorité sont traités avant ceux de plus forte priorité. Notez que le mot-clé `any` est doublé. Cela est nécessaire pour déléguer l'évaluation de la priorité (`priority`) de la file à un élément évaluable, *c.-à-d.* un arc parmi ceux présents dans ces listes. Le premier `any` délègue d'un niveau, *c.-à-d.* une liste, le second pointe vers un élément de cette liste, *c.-à-d.* un arc. Cet arc est donc le représentant de la file, et son propagateur est utilisé pour évaluer la file.

FIGURE 9.3 – Déclaration d'un moteur de propagation orienté variable.

```
1 : All :true;
2 : All as queue(wone) of {
3 :     each var as list(for)
4 : }
```

FIGURE 9.4 – Représentation sous la forme d'un arbre du moteur déclaré dans la Figure 9.3.



9.2 Évaluation des phases d'analyse grammaticale et de résolution

L'intérêt de la phase de prototypage est de construire et d'évaluer rapidement, mais sûrement, des moteurs de propagation. La phase d'analyse grammaticale n'est pas critique dans une démarche de prototypage, mais elle ne doit tout de même pas trop pénaliser les tests. La phase de résolution, quant à elle, est critique. Les versions interprétées des moteurs de propagation ne peuvent pas être compétitives avec les approches natives. Le coût d'utilisation doit donc être aussi faible que possible pour valider notre approche en tant qu'outil utilisable en pratique. Nous montrons que le surcoût induit est acceptable mais, surtout, que le classement des moteurs interprétés par rapport à leur temps d'exécution est généralement le même que le classement des moteurs natifs. La Table 9.4 reporte le surcoût de l'analyse grammaticale des fichiers MiniZinc avec la description des moteurs de propagation par rapport à l'analyse grammaticale du même fichier, sans la description, mais avec la création du moteur natif équivalent. La phase d'analyse grammaticale inclut le chargement du fichier, l'analyse lexicographique, la reconnaissance syntaxique et la traduction en instructions du solveur. La Table 9.4 reporte également le surcoût de résolution des instances avec les moteurs interprétés en comparaison avec les versions natives. Dans ce cas, le temps affiché exclut le temps d'analyse grammaticale. Cette étape est critique, la flexibilité qu'apporte notre approche ne doit pas être compromise par un coût d'utilisation rédhibitoire pour valider l'outil comme utile au prototypage, et utilisable en pratique.

Sans surprise, la phase d'analyse grammaticale souffre de l'ajout du DSL, et globalement cette phase requiert 28,46 % plus de temps, en moyenne (moyenne arithmétique des surcoûts). Les détails du nombre d'arcs créés sont donnés dans le Tableau 9.1, on peut constater le lien direct qui existe avec le temps d'analyse grammaticale : plus il y aura d'arcs, plus cette phase prendra de temps. Par exemple, c'est le cas pour les problèmes `slow_convergence`, `non_non_fast`, `latin-squares`, `debruijn_binary` et `bibd` pour lesquels le temps d'analyse syntaxique requiert *a minima* deux fois plus de temps.

TABLE 9.2 – Évaluation de la phase d'analyse grammaticale (temps en milli-secondes).

Problème	prop		var		7qd	
	natif	interprété	natif	interprété	natif	interprété
bacp-14	111.86	26.16%	109.88	26.99%	108.10	31.67%
bacp-15	109.58	28.95%	112.23	29.12%	108.15	27.90%
bibd_15_03_01	324.82	64.55%	324.94	63.92%	334.53	60.05%
black-hole_0	20.11	4.68%	19.94	7.10%	20.27	4.18%
black-hole_12	19.88	5.97%	21.40	5.02%	19.88	9.66%
CostasArray_14	25.93	22.43%	25.34	18.91%	25.91	20.94%
CostasArray_15	27.51	21.18%	27.43	20.67%	27.41	25.64%
debruijn_binary_02_08	156.50	38.80%	152.05	50.13%	157.66	21.56%
debruijn_binary_02_09	180.37	**72.11%**	190.39	64.18%	245.28	37.56%
fastfood_ff56	45.60	20.82%	46.69	15.77%	46.75	18.88%
fastfood_ff75	51.73	23.02%	52.10	20.13%	54.97	12.52%
fillomino_08	82.28	19.70%	81.85	19.32%	80.62	22.32%
fillomino_12	119.71	24.26%	118.61	26.93%	120.19	24.26%
ghoulomb_3-7-16	5451.97	39.91%	4974.15	56.90%	5327.58	43.26%
golomb_09	17.06	6.51%	16.94	10.65%	16.45	12.88%
golomb_10	18.91	8.70%	18.44	7.61%	18.43	11.68%
jobshop_jobshop_la03	65.66	22.56%	67.89	18.75%	62.94	29.66%
jobshop_jobshop_la18	94.68	29.07%	98.80	24.33%	102.56	20.60%
langford_l_2_09	49.61	16.86%	49.99	12.92%	50.29	14.87%
langford_l_2_10	55.18	20.85%	54.74	19.20%	56.90	20.48%
latin-squares-lp_12	69.19	45.33%	68.21	46.25%	71.41	45.66%
latin-squares-lp_15	112.73	52.77%	112.26	58.87%	112.62	56.78%
magicseq_030	99.96	19.73%	97.85	22.58%	95.01	24.67%
magicseq_040	146.75	24.00%	151.15	19.87%	145.73	25.64%
market_split_s4-04	15.07	7.10%	15.52	**-2.30%**	15.46	**-0.01%**
market_split_s4-10	15.02	**3.34%**	14.94	-1.55%	14.79	0.02%
non_non_fast_1	42.06	63.40%	44.00	**69.68%**	45.73	56.34%
non_non_fast_8	52.81	48.56%	54.00	55.89%	53.26	52.13%
p1f_10	48.10	39.99%	46.83	48.11%	47.12	48.20%
pentominoes-int_05	58.41	8.98%	61.63	-1.97%	59.11	10.33%
pentominoes-int_06	69.06	8.17%	68.08	7.05%	67.20	11.94%
radiation_04	52.01	23.37%	50.38	25.36%	50.20	30.03%
radiation_08	57.62	19.72%	56.85	23.67%	56.44	24.78%
rcpsp_max_psp_j10_33	222.06	31.61%	215.60	35.64%	218.56	33.67%
slow_convergence_0800	5444.76	54.35%	5375.88	50.33%	5473.01	57.28%
slow_convergence_0900	6850.97	52.64%	6509.14	55.18%	6553.78	**60.82%**
slow_convergence_1000	8137.86	70.46%	9120.74	41.07%	8613.91	56.30%
still_life_3x8	48.78	13.80%	47.90	18.94%	54.52	8.33%
still_life_5x5	50.29	9.44%	49.00	30.09%	48.72	21.12%
Moyenne		**28.56%**		**28.75%**		**28.07%**
Ecart-type		19.34		20.15		17.64

TABLE 9.3 – Évaluation de la phase de résolution (temps en secondes).

Problème	prop		var		7qd		rang
	natif	interprété	natif	interprété	natif	interprété	
bacp-14	13.39	16.62%	13.02	9.40%	17.31	22.21%	true
bacp-15	4.08	18.86%	3.97	12.57%	5.42	23.17%	true
bibd_15_03_01	11.86	4.72%	13.56	3.04%	10.46	8.95%	true
black-hole_0	14.87	2.52%	15.12	1.81%	13.90	0.33%	true
black-hole_12	2.43	−1.18%	2.58	1.08%	2.09	5.16%	true
CostasArray_14	1.08	11.24%	0.78	13.99%	1.01	16.59%	true
CostasArray_15	12.29	11.18%	8.80	10.04%	11.01	20.97%	true
debruijn_binary_02_08	0.70	**−6.09%**	0.61	9.41%	0.61	3.80%	true
debruijn_binary_02_09	2.47	3.48%	2.46	**−3.23%**	2.09	**−0.39%**	true
fastfood_ff56	8.35	12.79%	7.80	10.00%	7.05	15.73%	true
fastfood_ff75	16.14	21.57%	15.62	14.22%	16.07	15.74%	true
fillomino_08	3.62	16.39%	3.43	8.50%	2.38	21.86%	true
fillomino_12	14.66	11.45%	14.12	14.44%	9.03	19.70%	true
ghoulomb_3-7-16	14.98	19.87%	13.46	−1.32%	14.35	18.88%	true
golomb_09	0.53	7.26%	0.42	5.52%	0.43	14.86%	true
golomb_10	3.83	9.64%	2.99	4.85%	3.19	14.37%	true
jobshop_jobshop_la03	8.34	14.95%	19.03	15.09%	12.70	25.16%	true
jobshop_jobshop_la18	17.00	13.96%	16.98	11.37%	24.94	19.31%	true
langford_l_2_09	0.68	12.85%	0.57	12.02%	0.93	8.53%	true
langford_l_2_10	4.31	9.34%	3.51	12.87%	5.92	15.69%	true
latin-squares-lp_12	2.48	1.50%	2.43	3.69%	2.50	4.22%	true
latin-squares-lp_15	7.52	3.13%	7.35	3.95%	7.55	4.74%	true
magicseq_030	1.20	13.12%	0.98	11.65%	0.61	15.77%	true
magicseq_040	2.55	10.35%	2.48	6.28%	1.33	18.21%	true
market_split_s4-04	5.98	1.35%	5.62	8.37%	6.74	10.92%	true
market_split_s4-10	18.67	2.89%	17.26	10.72%	20.73	10.15%	true
non_non_fast_1	1.54	5.05%	1.60	4.01%	1.62	4.74%	true
non_non_fast_8	1.34	5.68%	1.36	2.11%	1.38	2.24%	true
p1f_10	4.27	7.78%	4.60	3.26%	4.22	10.32%	true
pentominoes-int_05	3.73	2.68%	3.74	1.57%	3.80	−0.13%	true
pentominoes-int_06	0.76	−0.29%	0.77	−1.04%	0.76	0.59%	true
radiation_04	1.36	10.09%	1.37	6.34%	1.54	14.75%	true
radiation_08	4.73	8.02%	4.90	5.91%	5.11	**26.88%**	true
rcpsp_max_psp_j10_33	15.80	18.83%	16.26	8.32%	20.32	23.82%	**false**
slow_convergence_0800	0.72	19.30%	0.70	**17.15%**	0.80	15.97%	true
slow_convergence_0900	0.91	11.32%	0.86	7.82%	1.05	13.92%	true
slow_convergence_1000	1.04	**27.55%**	0.95	9.36%	1.13	16.93%	true
still_life_3x8	0.94	1.80%	1.29	−2.98%	0.71	15.69%	true
still_life_5x5	0.63	2.36%	0.53	1.35%	0.44	14.32%	true
Moyenne		**9.33%**		**7.01%**		**13.20%**	
Écart-type		7.29		5.29		7.63	

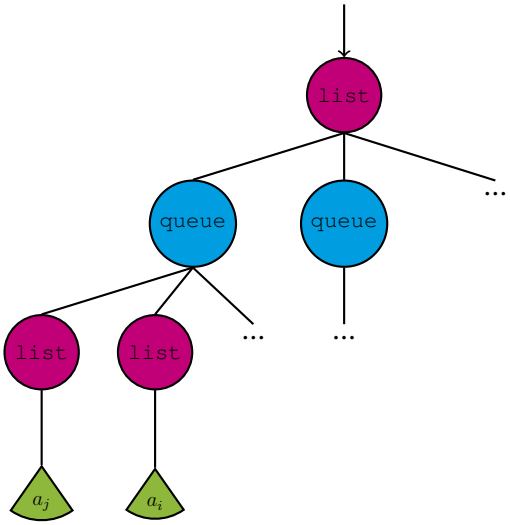
TABLE 9.4 – Tableau récapitulatif des surcoûts moyens induits par l'introduction du langage, sur 39 instances.

Moteur	Analyse grammaticale		Résolution	
	moyenne	écart-type	moyenne	écart-type
prop	28.56%	19.34	9.33%	7.29
var	28.75%	20.15	7.01%	5.29
7qd	28.07%	17.64	13.20%	7.63

FIGURE 9.5 – Déclaration d'un moteur de propagation orienté propagateur et priorité.

```
1 : All : true;
2 : All as list(wone) of {
3 :     each prop.prioDyn as queue(one) of {
4 :         each prop as list(for)
5 :     }key any.any.prop.prioDyn
6 : }
```

FIGURE 9.6 – Représentation sous la forme d'un arbre du moteur déclaré dans la Figure 9.5.



Toutefois, même si le surcoût semble important en apparence, le temps d'analyse grammaticale est négligeable par rapport à celui nécessaire à la résolution du problème (en deçà de 200 ms, sauf pour `slow_convergence`). De plus, ce surcoût vient non seulement de la nécessité de représenter tous les arcs en mémoire et de garantir les propriétés inhérentes au langage, mais également de la manière dont sont organisés les arcs. Par exemple, dans les instances où le nombre de variables est petit comparé au nombre de propagateurs, analyser la version interprétée de `var` se fait plus rapidement que pour `prop` et `7qd`. Enfin, on observe que parser `prop` et `7qd` est comparable, car regrouper les arcs autour des propagateurs est une opération commune aux deux déclarations.

Concernant la phase de résolution, nous pouvons observer que le surcoût moyen induit par l'utilisation d'un moteur interprété est de 9.81 % (moyenne arithmétique des surcoûts). De tels surcoûts proviennent vraisemblablement du manque d'optimisation dont souffrent les versions interprétées. Dans ces versions, lorsqu'un arc est planifié, la liste à laquelle il appartient doit être également planifiée, si nécessaire. Dans leurs versions natives, la seconde opération est naturellement supportée. Dans un langage de programmation qui supporte l'optimisation directe (tel que le langage C), les surcoûts doivent être plus facilement réduits. Cependant, le comportement de Java n'est pas parfaitement stable sur les très petites évaluations, dont le temps d'exécution est inférieur à la seconde. Le comportement de la JVM peut être bruité par les autres JVM précédemment lancées, mais également à cause de la compilation *juste à temps* et les optimisations liées au *Hot Spot* [80].

Les surcoûts d'utilisation liés à `var` et `prop` sont très proches de ceux des versions natives. Ce qui n'est pas le cas de `7qd`. Le langage introduit ici deux sources de faiblesse potentielles : une évaluation multicritères et des structures imbriquées. C'est particulière-

ment remarquable pour `7qd`. Dans sa version native, l'évaluation de `prioDyn` est faite de manière efficace en interrogeant directement le propagateur. Dans sa version interprétée, chaque arc doit être évalué à chaque planification. Donc, même si la version native de `7qd` est habituellement très performante, sa version interprétée est pénalisée. Sans surprise, une fois de plus, les moteurs basés sur le DSL induisent un surcoût. Cependant, il est intéressant de remarquer que dans 97.4 % des cas (38 instances sur les 39 présentes), le classement des moteurs par rapport à leur temps d'exécution est le même en version interprétée qu'en version native. Rappelons que les moteurs de propagation dans leur version native ont fait l'objet de multiples optimisations. Bien que les versions interprétées introduisent des surcoûts mémoire et temporel, ils assurent toutes les propriétés décrites dans la Section 8.4 naturellement. De plus, chaque déclaration peut être adaptée, enrichie en toute simplicité sans craindre d'introduire des erreurs ni avoir à se préoccuper de sa mise en œuvre. La prochaine étape d'évaluation concerne la mise en œuvre, sur un cas d'utilisation, du langage afin de démontrer la pertinence de son utilisation.

9.3 La règle de Golomb : un cas d'utilisation

Nous montrons maintenant, à l'aide d'un cas d'utilisation, combien il est facile de prototyper des moteurs de propagation grâce à notre langage dédié. Nous allons déclarer les moteurs de l'état de l'art et essayer deux autres moteurs adhoc.

Le problème du `Golomb ruler` (prob006, [41]) est défini comme "*un ensemble de* m *entiers* $0 = a_1 < a_2 < \cdots < a_m$ *tel que les* $\frac{m(m-1)}{2}$ *différences* $a_j - a_i, 1 \leq i < j \leq m$ *soient distinctes. Une telle règle contient* m *marques et est de taille* a_m*. L'objectif est de trouver une règle de taille minimum*".

En préparation des évaluations, nous avons instrumenté le modèle MiniZinc en ajoutant la déclaration des moteurs de propagation, et avons généré les fichiers FlatZinc correspondants, les uns avec $m = 10$ et les autres avec $m = 11$. Pour $m = 10$, en utilisant les contraintes globales disponibles dans la librairie du solveur de contraintes Choco, le problème est modélisé à l'aide de 55 variables, une contrainte `allDifferent`, 11 inégalités binaires et 45 équations ternaires. La stratégie de recherche est basée sur a : elle préserve l'ordre lexicographique et choisit la plus petite valeur de chaque variable choisie.

9.3.1 Description des moteurs

Tout d'abord, nous présentons les moteurs de propagation utilisés pour cette évaluation. En plus de `prop`, `var` et `7qd` présentés auparavant, nous déclarons `heap-var`, `heap-var-prio` et `2-coll`.

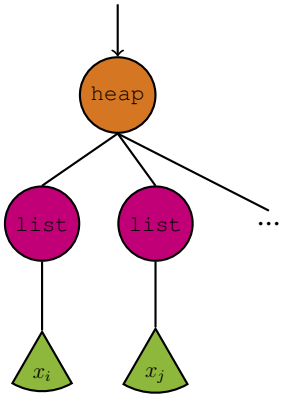
Premièrement, `heap-var` (Figure 9.7 et Figure 9.8) est un moteur orienté variables basé sur un tas. En effet, Boussemart et autres [19] ont montré que propager les événements en traitant d'abord ceux associés à la variable de plus petite cardinalité est une bonne stratégie pour améliorer la résolution de ce problème. Les arcs sont regroupés autour des variables dans des listes, ces listes sont ensuite ajoutées dans un tas. La liste qui contient la variable de plus petite cardinalité est alors sélectionnée pour être parcourue. Les arcs de cette liste sont ensuite propagés un à un une fois. Ces opérations sont répétées tant qu'il reste au moins un arc planifié dans une des listes.

Puis, nous déclarons `heap-var-prio` (Figure 9.9 et Figure 9.10), une variante de `heap-var` où les arcs, au sein de chaque liste, sont triés en fonction de la priorité de leur

FIGURE 9.7 – Déclaration d'un moteur de propagation basé sur un tas

```
1:All:true;
2:All as heap(wone) of {
3:     each var as list(for) key any.var.card
4:};
```

FIGURE 9.8 – Représentation sous la forme d'un arbre du moteur déclaré dans la Figure 9.7.



propagateur. L'idée ici est d'exécuter, pour une variable, les propagateurs par priorité croissante, *c.-à-d.* les plus légers d'abord. Cela devrait permettre de discriminer les propagateurs de la contrainte `AllDifferent` des autres.

Enfin, nous déclarons `2-coll` (Figure 9.11 et Figure 9.12), un moteur composé de deux collections. La première collection est basée sur `M`, l'ensemble des arcs associés aux variables a. Ces arcs sont placés dans une liste et triés par ordre lexicographique du nom de leur variable. Les événements intervenant sur la variable a_i seront traités avant ceux intervenant sur la variable a_{i+1}. L'itérateur `wfor` garantit, avec une propagation par *balayage*, d'atteindre un point fixe local (tous les événements des arcs de `M` auront été traités). La deuxième collection est une file qui stocke les arcs restants (ceux impliqués dans les variables de différences), elle garantit un second point fixe local (`wone`).

9.3.2 Évaluations des différents moteurs

La Table 9.5 montre les résultats obtenus en résolvant le problème de la règle de Golomb, pour $m = 10$ et $m = 11$, en utilisant différents moteurs interprétés décrits dans les Sections 9.1 et 9.3.1. Le temps de résolution (`time`, en secondes) et le nombre de propagations (`pex`) y sont reportés.

Il est difficile d'établir une relation directe entre le nombre de propagations et le temps de résolution. Par exemple, le moteur `7qd` demande 5.9 % plus de propagations que le moteur `prop`, cependant il nécessite 23.2 % moins de temps pour résoudre le problème. Cela s'explique par une meilleure répartition des propagations entre les contraintes légères et la contrainte globale `AllDifferent`, dont le coût de propagation est important mais qui est également moins souvent propagée (puisque de priorité plus faible que les autres propagateurs du modèle). Une sélection intelligente du prochain arc à propager est donc payante. Le même constat peut être fait concernant les moteurs `var` et `heap-var`. Nous observons les mêmes résultats que [19] : la sélection de la variable de plus petite cardinalité permet de résoudre plus efficacement le problème. Par contre, distinguer les propagateurs d'une variable

FIGURE 9.9 – Déclaration d'un second moteur de propagation basé sur un tas.

```
1 : All : true;
2 : All as heap(wone) of {
3 :     each var as list(for) of {
4 :          each prop.priority as list(for)
5 :          key any.prop.priority
6 :     } key any.any.var.card
7 : };
```

FIGURE 9.10 – Représentation sous la forme d'un arbre du moteur déclaré dans la Figure 9.9.

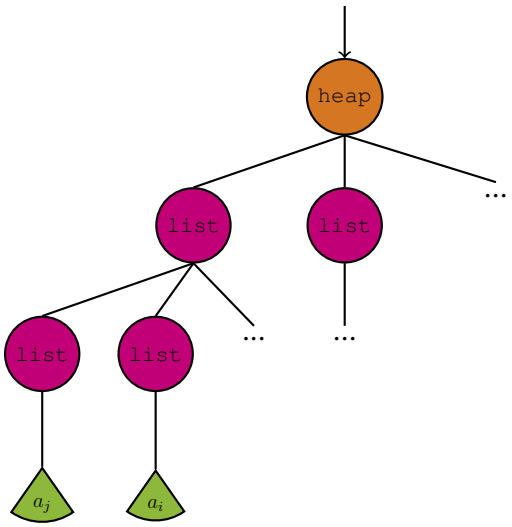


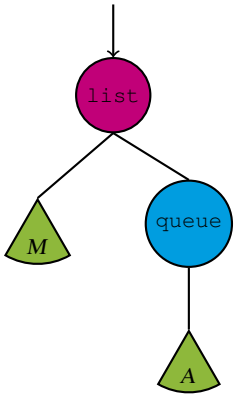
TABLE 9.5 – Moteurs évalués pour résoudre le problème de la règle de Golomb.

Engine	$m = 10$		$m = 11$	
	time (sec.)	pex (10^6)	time (sec.)	pex (10^6)
prop	3.83	10.085	76.70	191.894
var	2.99	10.305	55.50	192.034
7qd	3.19	10.860	60.67	200.108
heap-var	2.79	10.717	53.09	196.525
heap-var-prio	3.31	11.242	59.84	207.973
2-coll	**2.69**	**9.975**	**52.48**	**185.921**

FIGURE 9.11 – Déclaration d'un moteur de propagation constitué de deux collections.

```
1 : M : in(mark);
2 : A : true;
3 : list(wfor) of {
4 :   list(wfor) of {M key var.name},
5 :   queue(wone) of {A}
6 : };
```

FIGURE 9.12 – Représentation sous la forme d'un arbre du moteur déclaré dans la Figure 9.11.



grâce à la priorité, dans `heap-var-prio`, est contre-productif : cela retarde trop l'exécution des propagateurs de la contrainte `AllDifferent`, et donc des déductions moins précises sont propagées à chaque étape de propagation.

Enfin, le moteur `2-coll` est le plus efficace. Son comportement est comparable à celui du `heap-var`, mais il ne nécessite aucune évaluation dynamique de critères, ce qui le rend légèrement plus rapide. Il est basé sur une liste et une file dont les opérations se font en temps constant. Le nombre d'exécutions de propagateurs est également le plus faible. Ceci s'explique par la séparation qui est faite entre les variables a, d'une part, et les autres variables du modèle, d'autre part. En faisant ainsi, un sous-point fixe local peut être calculé sur un groupe de variables, avant de propager leurs modifications sur l'autre partie du modèle. Les propagations sont alors plus efficaces, puisque capitalisant au maximum les événements.

Ce qu'il est important que de remarquer à ce stade, c'est la facilité avec laquelle le prototypage a été effectué avec le DSL. Il a été possible, en quelques lignes, de définir une heuristique de propagation spécifique, reposant sur les propriétés des variables et des contraintes, et de produire un moteur de propagation sûr, interprétable et utilisable en pratique. Cela simplifie grandement le processus d'évaluation et promeut l'étude de l'ordre de révision des contraintes au sein de CSP.

10

Conclusion et travaux futurs

Notre première motivation, dans cette partie, était de fournir un outil qui rend possible la configuration du moteur de propagation dans un solveur de contraintes. Pour cela, nous avons proposé un langage dédié à la description des moteurs de propagation. Nous avons d'abord rappelé les techniques et services indispensables à l'implémentation du langage dans un solveur cible. Ensuite, nous avons présenté notre contribution : un langage dédié simple et évolutif qui permet aux utilisateurs avancés de décrire de manière concise mais puissante de moteurs de propagation. Les propriétés et garanties qu'il fournit ont ensuite été listées, recoupant non seulement les éléments liés au langage mais, également aux moteurs de propagation. Puis, nous avons présenté une implémentation dans le langage de modélisation MiniZinc et discuté des faiblesses de notre approche. Enfin, nous avons évalué les surcoûts liés à l'utilisation du langage lors des phases d'analyse grammaticale et de résolution. Nous avons montré l'expressivité de notre proposition et la flexibilité qu'elle permet d'apporter aux solveurs. En plus des moteurs communément utilisés, notre langage permet la déclaration de schémas de propagation très précis. Cette étude a fait l'objet de publication aux Journées Francophone de Programmation par Contraintes [92] et dans le journal Constraints [93].

Il existe bien entendu des limites à la configuration des moteurs de propagation par notre approche. La première, la plus importante, mais également la plus problématique à mettre en œuvre, concerne la prise en compte de la décomposition des contraintes, et surtout son impact sur les moteurs de propagation produits. La restriction majeure réside dans les prédicats et attributs nécessaires à la définition des groupes ; des alternatives doivent être proposées. Nous avons suggéré, en calquant le fonctionnement de MiniZinc concernant les contraintes non supportées par un solveur cible, de définir un schéma de propagation type par contrainte globale reformulée. Bien entendu, un tel choix devrait être évalué pour juger de l'interaction des schémas de décomposition sur la propagation. Cependant, l'impact même des décompositions sur la propagation n'a jamais été véritablement mesuré, on s'intéresse souvent en premier au niveau de filtrage d'une décomposition. La seconde limite qui doit être considérée est inhérente à l'utilisation même d'un langage dédié et concerne la création des moteurs de propagation "mal formés" dont il est important d'empêcher la création. Une approche envisageable est d'établir des règles de transformations et d'optimisation des descriptions qui ne dénaturent pas la description initiale, mais la rendent plus compacte et donc évaluable plus efficacement. Notre DSL pourrait alors faire partie intégrante des standards pour la pro-

grammation par contraintes (MiniZinc [36], JSR331 [35]). Dans un souci de complétude, nous pourrions également enrichir le langage pour prendre en compte la *programmation automatique*, technique subsidiaire à la propagation de contrainte, présentée par Laurière [65], qui consiste à remplacer la propagation par la génération d'un programme d'énumération des solutions, dès lors que la propagation devient chère et inefficace. L'approche que nous proposons, basée sur un langage dédiée, repose sur les connaissances de l'utilisateur. Sa maitrise du modèle déclaré et son expérience doivent lui permettre de corriger les pathologies de propagation, mal traitées par les moteurs proposés par défaut. D'autre part, il est intéressant de constater que, à l'heure où les solveurs à génération paresseuse de clauses [79] font de plus en plus parler d'eux, la recherche n'apparait plus comme aussi importante[1]. Dans ces outils, la propagation peut alors redevenir critique. De la même manière, dans une recherche à voisinage large, où la faculté d'explorer beaucoup de voisinages par seconde est gage d'efficacité, travailler sur la propagation des contraintes peut sans doute faire l'objet d'une étude complémentaire à la conception des voisinages.

En marge de ce travail sur le langage dédié à la propagation, nous nous sommes intéressés à deux autres aspects de la propagation : le changement *à la volée* de moteur de propagation, d'une part, et une configuration automatique du moteur, d'autre part. Le changement de moteur que nous qualifions d'*à la volée* consiste à choisir, à partir d'un portefeuille de moteurs, celui qui s'adapte le mieux à la situation. Nous avons par exemple distingué le moteur à utiliser lors de la propagation initiale de celui utilisé pour la résolution du problème[2]. Nous avons obtenu quelques résultats intéressants, notamment concernant le problème de la séquence magique où basculer d'un moteur orienté propagateurs à un moteur orienté variables permet d'obtenir des gains. Toutefois, cette approche doit être confirmée sur d'autres problèmes.Concernant la configuration automatique, nous nous sommes inspirés des travaux concernant la stratégie de recherche et plus particulièrement ceux concernant l'adaptation des choix de branchement en observant l'*activité* des décisions [76]. Nous avons évalué un moteur de propagation basé sur critère dynamique ayant attrait à la propagation des contraintes. Cette approche mesure l'impact d'un propagation en terme de variables modifiées pour faciliter le choix du prochain propagateur à exécuter. Bien que les résultats préliminaires soient encourageants, d'autres critères discriminants doivent être envisagées et un travail sur la structure de données stockant cette information doit être effectué. En effet, une tolérance à l'erreur est certainement acceptable et une approche basée sur un filtre de Bloom [17] pourrait clairement en améliorer les performances.

En conclusion, il nous apparait bon de rappeler que le travail sur les moteurs de propagation a donné lieu à une nouvelle version du solveur de contraintes Choco [90, 91]. Cette nouvelle version a été réimplémentée entièrement, et le travail effectué sur les moteurs de propagation a clairement bénéficié à la totalité du solveur. Chacune des contraintes portées depuis la précédente version du solveur a été durement testée dans plusieurs moteurs de propagation, permettant de les rendre plus efficaces mais, surtout, plus fiables. Concrètement, cela s'est traduit par une amélioration du classement de Choco dans le Challenge MiniZinc entre 2012 et 2013, toutes catégories de recherche confondues. On notera, à ce propos, que Choco s'est classé second dans les catégories *Parallel search* et *Open class*[3], juste derrière or-tools [82].

[1] Peter J. Stuckey affirme même que "search is dead" (Peter J. Stuckey, 2013,"Those who cannot remember the past are condemned to repeat it", Invited Talk, CP'13).

[2] Dans Choco, la propagation initiale est distinguée de l'étape de résolution.

[3] http://www.minizinc.org/challenge2013/results2013.html

IV

Conclusion

11

Contributions

Dans cette thèse, nous nous sommes intéressés à la conception d'un solveur de contraintes dans le but d'en améliorer l'utilisation et ainsi poursuivre la connexion existant entre l'utopie de *boîte noire*, chère à Freuder, et la réalité, des outils flexibles, typés *boite blanche*. Dans ce contexte, nos résultats démontrent, si c'était encore nécessaire, que ces deux aspects de la programmation par contraintes peuvent coexister, et que cette cohabitation permet de proposer le même outil qui répond aux attentes d'un utilisateur novice et d'un modeleur expérimenté.

Notre apport à la transversalité des solveurs de contraintes est double :

- Un système d'explications est un socle souvent négligé utile à la conception d'outils *boîte noire*. C'est le cas particulièrement pour la résolution de problèmes d'optimisation où construire une solution de bonne qualité est un premier souhait des modeleurs. Les informations que fournissent les explications permettent de concevoir des voisinages génériques et efficaces pour la recherche à voisinage large, enrichissant le portefolio d'heuristiques adaptatives disponibles. Nous avons ainsi proposé un système de calcul d'explications paresseuses, dont la compilation n'intervient que lors d'obtention de solutions, et conçu deux voisinages dont l'algorithme exploite cette base d'explications. Ceux-ci répondent aux questions suivantes : comment guider le solveur hors d'une zone d'échec lié à la coupe ? Et comment l'aider à trouver des solution de meilleure qualité ? Les diverses combinaisons de voisinages que nous avons proposé se sont montrées très compétitives avec la solution générique proposé par Perron et autres. Dans une certaine mesure, nos travaux s'inscrivent dans la continuité de ceux d'Hadrien Cambazard : « les explications sont sous-exploitées à l'heure actuelle (...) elles offrent de nombreuses perspectives qui ne passent pas forcément par le backtrack intelligent pour la résolution elle-même » (Hadrien Cambazard, 2006, "Résolution de problèmes combinatoires par des approches fondées sur la notion d'explication", [22]).

- Dans une démarche d'adaptation d'un solveur au problème qu'il doit résoudre, le contrôle de l'ordonnancement des contraintes à propager reste, de manière générale, inexploité dans les solveurs actuels. Ce constat n'est pas uniquement imputable à la faible espérance de gain relative à la propagation (l'algorithme est polynomial), mais également à la pénibilité liée à la mise en œuvre des moteurs de propagation. Pour

répondre à ce deuxième point, nous avons proposé un langage dédié opérationnel, qui offre des concepts de haut niveau permettant à l'utilisateur d'ignorer les détails d'implémentation du solveur, tout en conservant un bon niveau de flexibilité et des garanties. Il permet à l'utilisateur expérimenté, et au développeur, de décrire des schémas de propagation des contraintes adaptés aux problèmes traités, et ainsi de prototyper et d'évaluer rapidement ces schémas. Ce langage, et son implémentation, vient en complément des techniques largement répandues dans les solveurs actuels.

Ces études constituent des contributions originales pour ces deux thématiques et ont fait l'objet de publications [92, 93, 94, 95]. Elles ont été évaluées dans le solveur de contraintes Choco [90, 91], et sont disponibles en ligne. Cette nouvelle version du solveur est le fruit des réflexions qui ont menées autour de la conception de solveurs, et plus particulièrement sur les deux axes que nous avons présentés dans cette thèse.

Enfin, ces dernières années, une volonté de vulgarisation de la programmation par contraintes se dessine. Cela se traduit, d'une part, par un regain d'intérêt pour les langages de modélisation, comme MiniZinc [36], et, d'autre part, par la mise en place d'une standardisation des outils, via le JSR331 [35]. Un accroissement du nombre de solveurs de contraintes développés au sein de la communauté PPC accompagne ce phénomène. La compétitivité qui en résulte, notamment par le biais du Challenge MiniZinc, force les développeurs des solveurs à concevoir des outils efficaces mais flexibles, conceptuellement typés *boîte blanche*, qui répondent à une philosophie particulière, mais qui peuvent également fonctionner en mode *boîte noire*, pour s'interfacer avec les langages de modélisation. C'est là toute l'ambiguité de la programmation par contraintes, mais qui en fait également sa richesse. Nos contributions se situent dans ce schéma, en tentant de joindre ces deux facettes dans le solveur Choco.

À l'origine de cette thèse, il y a le souhait de concevoir un outil de programmation par contraintes qui soit efficace, accessible au plus grand nombre mais dont chacun des mécanismes puisse être pris en main par celui qui le désire. Le principal verrou de la version précédente du solveur de contraintes Choco se situait dans la rigidité de son moteur de propagation. Nous étions, assez naïvement sans doute, persuadés de l'immense plus-value qu'une adaptation du schéma de propagation au problème traité pouvait apporter. Cependant, sans prétention, cette thèse et le travail d'ingénierie logicielle qu'elle a nécessité dans le solveur Choco, aura conduit aux deux contributions suivantes. D'une part, la mise à disposition d'un solveur *open source* nativement expliqué, prenant le relais du solveur PaLM [52]. Les explications suscitent un regain d'intérêt dans la communauté, grâce aux solveurs à génération paresseux de clauses, et nous pensons qu'il y a encore beaucoup de publicité à faire autour de cette technique, et qui aura un rôle important à jouer dans les années à suivre. D'autre part, en collaboration avec Laurent Perron, nous avons contribué à relancer l'intérêt du Challenge MiniZinc, en bousculant l'ordre établi depuis 2008. En ce sens, nous pouvons nous targuer d'avoir également contribué à l'amélioration du solveur Gecode [109].

12

Perspectives

Les interactions entre les différents éléments constitutifs d'un solveur sont multiples et, le plus souvent, difficile à mesurer. C'est pourquoi dans les études menées au cours de cette thèse nous avons délibérément restreint la déclarativité de l'outil, en nous interdisant d'utiliser des contraintes globales. Un tel choix doit, bien sûr, être reconsidéré dans le futur. Les contraintes globales, de part leur nature, contribuent à l'amélioration du traitement des problèmes, au niveau sémantique, opérationnel ou algorithmique. Limplémentation de leurs algorithmes de filtrage nécessitent de prendre en compte les spécificités du solveur cible, en particulier celles du moteur de propagation. Une première perspective est d'étudier le comportement d'une décomposition par rapport à la contrainte référence, du point de vue de la propagation. Il s'agit de trouver un juste équilibre entre le rendement effectif, en terme de filtrage, et l'efficacité, en terme de temps de calcul. À ce jour, l'impact du moteur de propagation sur l'efficacité d'un propagateur est difficilement mesurable, d'autant plus que la manière dont les événements sont orchestrés est souvent ignorée lors de l'étude des algorithmes de filtrage des contraintes.

L'interdépendance des opérations joue également un rôle dans le calcul des explications : l'ordre dans lequel sont modifiées les variables influe sur la pertinence des explications produites. L'étude de l'influence des décompositions de contraintes sur les explications constitue une seconde perspective. En effet, à ce jour, peu de contraintes globales sont formellement expliquées, ce qui n'encourage pas les développeurs de solveurs à enrichir leurs outils avec des systèmes d'explications. Un aspect de cette étude pourrait être d'expliquer des contraintes globales complexes à l'aide d'une ou plusieurs de leurs décompositions. Les schémas d'explications en seraient alors simplifiés, et la recherche jouirait, tout de même, des avantages liés à la nature globale des contraintes.

Un troisième axe d'étude se dessine autour des besoins, en terme de propagation, de techniques comme la recherche à large voisinage. Cette méthode ne repose plus que partiellement sur les stratégies de recherche arborescente, puisqu'une partie des variables est déjà instanciée quand l'exploration est lancée. En ce sens, elle repose un peu plus sur les performances de l'algorithme de propagation des contraintes. En effet, la recherche à voisinage large se doit d'effectuer le plus de "mouvements" possibles dans le laps de temps qui lui est alloué pour résoudre un problème. Des techniques comme l'adaptation du schéma de propagation aux voisins générés ou bien la programmation automatique doivent être envisagées.

Elles devraient permettre de s'acquitter d'un algorithme de propagation générique, bon en moyenne, pour satisfaire un besoin d'efficacité et de spécificité.

Un dernier axe d'étude s'articule autour des solveurs à génération paresseuse de clauses. Ces outils hybrides apparaissent clairement comme des alternatives sérieuses aux solveurs de contraintes « traditionnels ». Il est manifeste qu'ils peuvent bénéficier des travaux décrits dans cette thèse. À notre connaissance, aucune étude n'a été faite quant à l'utilisation de tels solveurs avec une recherche locale à voisinage large. Pourtant, comme ces outils utilisent déjà les explications pour améliorer le parcours de l'espace de recherche, ils bénéficieraient pleinement, et à développement limité, des voisinages que nous avons proposés.

Enfin, d'un point de vue logiciel, nous avons pour objectif de continuer à proposer un outil à destination de tous, et notamment des industriels. L'une des spécificités des problèmes qu'ils formalisent est qu'ils reposent sur des données incertaines : certains domaines sont corrigés, les caractéristiques des problèmes évoluent en fonction des besoins ou de la réglementation, etc. Ces problèmes sont intrinsèquement dynamiques, et les solutions logicielles doivent apprivoiser cette instabilité pour pouvoir les traiter opérationnellement. Nous pensons que les méthodes étudiés dans cette thèse constituent une première réponse à cette problématique. En réinterprétant la notion de voisinage, les techniques de recherche locale permettent de prendre en compte les incertitudes. Délimiter le périmètre de leurs portées sur le modèle peut se faire à l'aide de la propagation des contraintes. Et enfin, les explications sont en mesure de fournir des suggestions de réparations opportunes.

Bibliographie

[1] Abderrahmane Aggoun and Nicolas Beldiceanu. Time stamps techniques for the trailed data in constraint logic programming systems. In *SPLT*, pages 487–510, 1990. 32

[2] Kendall E. Atkins. An introduction to numerical analysis, 1989. 65

[3] John W. Backus, Friedrich L. Bauer, Julien Green, C. Katz, John McCarthy, Alan J. Perlis, Heinz Rutishauser, Klaus Samelson, Bernard Vauquois, Joseph Henry Wegstein, Adriaan van Wijngaarden, Michael Woodger, and Peter Naur. Revised report on the algorithm language algol 60. *Commun. ACM*, 6(1) :1–17, 1963. 91

[4] Amine Balafrej, Christian Bessiere, Remi Coletta, and El-Houssine Bouyakhf. Adaptive parameterized consistency. In *CP*, pages 143–158, 2013. 24

[5] J. Christopher Beck and Laurent Perron. Discrepancy-bounded depth first search. In *In Proc. of CP-AI-OR ?2000*, pages 8–10, 2000. 33

[6] Nicolas Beldiceanu, Mats Carlsson, Sophie Demassey, and Thierry Petit. Global constraint catalogue : Past, present and future. *Constraints*, 12(1) :21–62, 2007. 26

[7] Christian Bessière. Arc-consistency and arc-consistency again. *Artif. Intell.*, 65(1) :179–190, 1994. 24

[8] Christian Bessière. Constraint propagation. Technical Report 06020, Laboratoire d ?Informatique, de Robotique et de Microélectronique de Montpellier, 2006. 19, 20, 21, 86

[9] Christian Bessière and Marie-Odile Cordier. Arc-consistency and arc-consistency again. In *AAAI*, pages 108–113, 1993. 24

[10] Christian Bessière, Eugene C. Freuder, and Jean-Charles Régin. Using inference to reduce arc consistency computation. In *IJCAI*, pages 592–599, 1995. 24

[11] Christian Bessière, Eugene C. Freuder, and Jean-Charles Régin. Using constraint metaknowledge to reduce arc consistency computation. *Artif. Intell.*, 107(1) :125–148, 1999. 24

[12] Christian Bessière and Pascal Van Hentenryck. To be or not to be ... a global constraint. In *CP*, pages 789–794, 2003. 25

[13] Christian Bessière and Jean-Charles Régin. Arc consistency for general constraint networks : Preliminary results. In *IJCAI (1)*, pages 398–404, 1997. 24

[14] Christian Bessière and Jean-Charles Régin. Refining the basic constraint propagation algorithm. In *IJCAI*, pages 309–315, 2001. 24

[15] Christian Bessière, Jean-Charles Régin, Roland H. C. Yap, and Yuanlin Zhang. An optimal coarse-grained arc consistency algorithm. *Artif. Intell.*, 165(2) :165–185, 2005. 24

[16] Christian Bessiere, Kostas Stergiou, and Toby Walsh. Domain filtering consistencies for non-binary constraints. *Artif. Intell.*, 172(6-7) :800–822, 2008. 24

[17] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7) :422–426, July 1970. 118

[18] Jérémie Du Boisberranger, Danièle Gardy, Xavier Lorca, and Charlotte Truchet. When is it worthwhile to propagate a constraint ? a probabilistic analysis of alldifferent. In *ANALCO*, pages 80–90, 2013. 30

[19] Frédéric Boussemart, Fred Hemery, and Christophe Lecoutre. Revision ordering heuristics for the constraint satisfaction problem. In *1st International Workshop on Constraint Propagation and Implementation held with CP'04(CPAI'04)*, pages 9–43, Toronto, Canada, sep 2004. 29, 30, 90, 112, 113

[20] Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting systematic search by weighting constraints. In Ramon López de Mántaras and Lorenza Saitta, editors, *ECAI*, pages 146–150. IOS Press, 2004. 14

[21] Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Mouny Samy-Modeliar. Contrôle statistique du processus de propagation de contraintes. In *7ièmes Journées Francophones de Programmation par Contraintes (JFPC'11)*, pages 65–74, Lyon, France, jun 2011. 30, 99

[22] Hadrien Cambazard. *Résolution de problèmes combinatoires par des approches fondées sur la notion d ?explication*. PhD thesis, Université de Nantes, 2006. 121

[23] Hadrien Cambazard and Narendra Jussien. Identifying and exploiting problem structures using explanation-based constraint programming. *Constraints*, 11(4) :295–313, 2006. 48

[24] Mats Carlsson, Greger Ottosson, and Björn Carlson. An open-ended finite domain constraint solver. In Hugh Glaser, Pieter H. Hartel, and Herbert Kuchen, editors, *PLILP*, volume 1292 of *Lecture Notes in Computer Science*, pages 191–206. Springer, 1997. 28, 86

[25] Alain Chabrier, Emilie Danna, Claude Le Pape, and Laurent Perron. Solving a network design problem. *Annals of Operations Research*, 130(1-4) :217–239, 2004. 35

[26] Assef Chmeiss and Philippe Jégou. Efficient path-consistency propagation. *International Journal on Artificial Intelligence Tools*, 7(2) :121–142, 1998. 24

[27] Nicos Christofides, Aristide Mingozzi, and Paolo Toth. The vehicle routing problem. chapter 11, pages 315–338. 1979. 35

[28] Geoffrey G. Chu. *Improving combinatorial optimization*. PhD thesis, The University of Melbourne, 2011. 42

[29] Pedro J Copado-Méndez, Christian Blum, Gonzalo Guillén-Gosálbez, and Laureano Jiménez. Application of large neighborhood search to strategic supply chain management in the chemical industry. In *Hybrid Metaheuristics*, pages 335–352. Springer, 2013. 36

[30] Emilie Danna and Laurent Perron. Structured vs. unstructured large neighborhood search : A case study on job-shop scheduling problems with earliness and tardiness costs. In Francesca Rossi, editor, *Principles and Practice of Constraint Programming, CP 2003*, volume 2833 of *Lecture Notes in Computer Science*, pages 817–821. Springer Berlin Heidelberg, 2003. 34, 35

[31] Romuald Debruyne and Christian Bessière. Domain filtering consistencies. *J. Artif. Intell. Res. (JAIR)*, 14 :205–230, 2001. 24

[32] Romuald Debruyne, Gérard Ferrand, Narendra Jussien, Willy Lesaint, Samir Ouis, and Alexandre Tessier. Correctness of constraint retraction algorithms. In *FLAIRS'03 : Sixteenth international Florida Artificial Intelligence Research Society conference*, pages 172–176, St. Augustine, Florida, USA, May 2003. AAAI press. 37, 42

[33] Rina Dechter. Enhancement schemes for constraint processing : Backjumping, learning, and cutset decomposition. *Artif. Intell.*, 41(3) :273–312, 1990. 37, 38

[34] Emrah Demir, Tolga Bektaş, and Gilbert Laporte. An adaptive large neighborhood search heuristic for the pollution-routing problem. *European Journal of Operational Research*, 2012. 36

[35] Jacob Feldman. Java Specification Request 331 : Constraint Programming API. `http://jcp.org/en/jsr/detail?id=331`. Accès : 13 Novembre 2013. 118, 122

[36] G12. MiniZinc and FlatZinc. `http://www.minizinc.org/`. Accès : 12 Novembre 2013. 49, 87, 89, 100, 105, 118, 122

[37] Ian P. Gent, Christopher Jefferson, and Ian Miguel. Minion : A fast scalable constraint solver. In *ECAI*, pages 98–102, 2006. 28, 30, 86

[38] Ian P. Gent, Christopher Jefferson, and Ian Miguel. Watched literals for constraint propagation in minion. In *CP*, pages 182–197, 2006. 28, 30, 90

[39] Ian P. Gent, Ian Miguel, and NeilC.A. Moore. Lazy explanations for constraint propagators. In Manuel Carro and Ricardo Peña, editors, *Practical Aspects of Declarative Languages*, volume 5937 of *Lecture Notes in Computer Science*, pages 217–233. Springer Berlin Heidelberg, 2010. 58

[40] Ian P. Gent, Ian Miguel, and Peter Nightingale. Generalised arc consistency for the alldifferent constraint : An empirical survey. *Artif. Intell.*, 172(18) :1973–2000, 2008. 29

[41] Ian P. Gent and Toby Walsh. CSPLib : a problem library for constraints. `http://www.csplib.org/`. Accès : 12 Novembre 2013. 100, 112

[42] M. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1 :25–46, 1993. 37, 39, 58

[43] Daniel Godard, Philippe Laborie, and Wim Nuijten. Randomized large neighborhood search for cumulative scheduling. In *ICAPS*, volume 5, pages 81–89, 2005. 35

[44] Michel Gondran, Michel Minoux, and Steven Vajda. *Graphs and Algorithms*. John Wiley & Sons, Inc., New York, NY, USA, 1984. 26

[45] Robert M. Haralick and Gordon L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. In *Proceedings of the 6th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'79, pages 356–364, San Francisco, CA, USA, 1979. Morgan Kaufmann Publishers Inc. 48

[46] Warwick Harvey and Joachim Schimpf. Bounds consistency techniques for long linear constraints, 2002. 54

[47] William D. Harvey and Matthew L. Ginsberg. Limited discrepancy search. In *IJCAI (1)*, pages 607–615, 1995. 33

[48] Pascal Van Hentenryck, Yves Deville, and Choh-Man Teng. A generic arc-consistency algorithm and its specializations. *Artif. Intell.*, 57(2-3) :291–321, 1992. 24, 86

[49] ILOG and IBM. IBM ILOG CPLEX CP Optimizer. `http://www-01.ibm.com/software/commerce/optimization/cplex-cp-optimizer/index.html`. Accès : 8 Novembre 2013. 14, 85, 86

[50] N. Jussien and O. Lhomme. Local search with constraint propagation and conflict-based heuristics. *Artificial Intelligence*, 139(1) :21–45, July 2002. 37, 39, 58

[51] Narendra Jussien. The versatility of using explanations within constraint programming. Technical Report 03-04-INFO, École des Mines de Nantes, 2003. 39, 47, 54, 58

[52] Narendra Jussien and Vincent Barichard. The palm system : explanation-based constraint programming. In *In Proceedings of TRICS : Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP 2000*, pages 118–133, 2000. 82, 122

[53] Narendra Jussien, Romuald Debruyne, and Patrice Boizumault. Maintaining arc-consistency within dynamic backtracking. In *Principles and Practice of Constraint Programming (CP 2000)*, number 1894 in Lecture Notes in Computer Science, pages 249–261, Singapore, September 2000. Springer-Verlag. 39

[54] Narendra Jussien and Olivier Lhomme. Dynamic domain splitting for numeric csps. In *European Conference on Artificial Intelligence (ECAI'98)*, pages 224–228, 1998. 56, 59

[55] Irit Katriel. Expected-case analysis for delayed filtering. In *CPAIOR*, pages 119–125, 2006. 99

[56] George Katsirelos and Fahiem Bacchus. Generalized nogoods in csps. In *AAAI*, pages 390–396, 2005. 37

[57] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *SCIENCE*, 220(4598) :671–680, 1983. 35

[58] Richard E. Korf. Improved limited discrepancy search. In *AAAI/IAAI, Vol. 1*, pages 286–291, 1996. 33

[59] Attila A Kovacs, Sophie N Parragh, Karl F Doerner, and Richard F Hartl. Adaptive large neighborhood search for service technician routing and scheduling problems. *Journal of scheduling*, 15(5) :579–600, 2012. 36

[60] Krzysztof Kuchcinski and Rados ?aw Szymanek. JaCoP : Java Constraint Solver. `http://www.jacop.eu/`. Accès : 15 Novembre 2013. 86

[61] P. Laborie and D. Godard. Self-adapting large neighborhood search :application to single-mode scheduling problems. In P. Baptiste, G. Kendall, A. Munier-Kordon, and F. Sourd, editors, *In proceedings of the 3rd Multidisciplinary International Conference on Scheduling : Theory and Applications (MISTA 2007), 28 -31 August 2007, Paris, France*, pages 276–284, 2007. Paper. 35

[62] Francois Laburthe and le projet OCRE. Choco : implémentation du noyau d'un système de contraintes. In *Actes des 6e Journées Nationales sur la résolution de Problèmes NP-Complets*, pages 151–165, June 2000. 28, 29, 30, 86

[63] Mikael Z. Lagerkvist and Christian Schulte. Advisors for incremental propagation. In Christian Bessiere, editor, *CP*, volume 4741 of *Lecture Notes in Computer Science*, pages 409–422. Springer, 2007. 29, 30, 90

[64] Mikael Z. Lagerkvist and Christian Schulte. Propagator groups. In Ian P. Gent, editor, *CP*, volume 5732 of *Lecture Notes in Computer Science*, pages 524–538. Springer, 2009. 29, 30, 90, 91

[65] Jean-Louis Laurière. Constraint propagation or automatic programming. 1976. 118

[66] Christophe Lecoutre, Frédéric Boussemart, and Fred Hemery. Exploiting multidirectionality in coarse-grained arc consistency algorithms. In *CP*, pages 480–494, 2003. 24

[67] Christophe Lecoutre, Lakhdar Sais, Sébastien Tabary, and Vincent Vidal. Recording and minimizing nogoods from restarts. *JSAT*, 1(3-4) :147–167, 2007. 82

[68] Alejandro López-Ortiz, Claude-Guy Quimper, John Tromp, and Peter van Beek. A fast and simple algorithm for bounds consistency of the alldifferent constraint. In Georg Gottlob and Toby Walsh, editors, *IJCAI*, pages 245–250. Morgan Kaufmann, 2003. 25, 26

[69] Alan Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1) :99–118, 1977. 21, 86

[70] Jean-Baptiste Mairy, Yves Deville, and Pascal Van Hentenryck. Reinforced adaptive large neighborhood search. In *8th Workshop on Local Search techniques in Constraint Satisfaction (LSCS2011)*, 2011. 35

[71] Jean-Baptiste Mairy, Pierre Schaus, and Yves Deville. Generic adaptive heuristics for large neighborhood search. In *7th Workshop on Local Search Techniques in Constraint Satisfaction (LSCS2010)*, 2010. 36, 47

[72] Arnaud Malapert and Jean-Charles Régin. Résolution efficace de problèmes simples mais de grande taille. In Simon de Givry, editor, *Huitièmes Journées Francophones de Programmation par Contraintes - JFPC 2012*, Toulouse, France, May 2012. 86

[73] Yuri Malitsky, Deepak Mehta, Barry O'Sullivan, and Helmut Simonis. Tuning parameters of large neighborhood search for the machine reassignment problem. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 176–192. Springer, 2013. 36

[74] Joaäo P. Marques-silva and Karem A. Sakallah. Grasp : A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48 :506–521, 1999. 43

[75] J. J. McGregor. Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. *Inf. Sci.*, 19(3) :229–250, 1979. 23, 28, 86

[76] Laurent Michel and Pascal Van Hentenryck. Activity-based search for black-box constraint programming solvers. In *CPAIOR*, pages 228–243, 2012. 14, 118

[77] Roger Mohr and Thomas C. Henderson. Arc and path consistency revisited. *Artif. Intell.*, 28(2) :225–233, 1986. 23

[78] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff : Engineering an efficient sat solver. In *DAC*, pages 530–535, 2001. 14, 28, 37, 42, 43

[79] Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation via lazy clause generation. *Constraints*, 14(3) :357–391, 2009. 42, 48, 118

[80] Oracle. Java SE 6 Performance White Paper. `http://www.oracle.com/technetwork/java/6-performance-137236.html`. Accès : 12 Novembre 2013. 111

[81] Terrence Parr. ANTLR v3. `http://www.antlr3.org/`. Accès : 12 Novembre 2013. 100

[82] Laurent Perron. or-tools : Operations Research Tools developed at Google. `http://code.google.com/p/or-tools/`. Accès : 15 Novembre 2013. 86, 118

[83] Laurent Perron. Fast restart policies and large neighborhood search. In Francesca Rossi, editor, *Principles and Practice of Constraint Programming at CP 2003*, volume 2833 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2003. 36, 62

[84] Laurent Perron and Paul Shaw. Combining forces to solve the car sequencing problem. In Jean-Charles Régin and Michel Rueher, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 3011 of *Lecture Notes in Computer Science*, pages 225–239. Springer Berlin Heidelberg, 2004. 35

[85] Laurent Perron, Paul Shaw, and Vincent Furnon. Propagation guided large neighborhood search. In *CP'04*, pages 468–481, 2004. 36, 47, 49, 61, 62, 63, 75

[86] David Pisinger and Stefan Ropke. Large neighborhood search. In *Handbook of metaheuristics*, pages 399–419. Springer, 2010. 34, 47

[87] Cédric Pralet and Gérard Verfaillie. Travelling in the world of local searches in the space of partial assignments. In *CPAIOR*, pages 240–255, 2004. 39

[88] Cédric Pralet and Gérard Verfaillie. About the choice of the variable to unassign in a decision repair algorithm. *RAIRO - Operations Research*, 39(1) :55–74, 2005. 39

[89] Patrick Prosser. MAC-CBJ : maintaining arc consistency with conflict-directed backjumping. Technical Report 95/177, University of Strathclyde, Dept. of Computer Science, University of Strathclyde, 1995. 37, 38, 58

[90] Charles Prud'homme and Jean-Guillaume Fages. Introduction to choco3. In *1st Workshop on CPSolvers : Modeling, Applications, Integration, and Standardization, CP 2013*, 2013. 15, 49, 62, 82, 105, 118, 122

[91] Charles Prud'homme, Jean-Guillaume Fages, Xavier Lorca, and Narendra Jussien. Choco-3.1.0. `http://www.emn.fr/z-info/choco-solver/index.php?page=choco-3`. Accès : 15 Novembre 2013. 15, 49, 62, 82, 86, 105, 118, 122

[92] Charles Prud'homme, Xavier Lorca, Rémi Douence, and Narendra Jussien. Prototyper des moteurs de propagation avec un dsl. In *9èmes Journées Francophones de Programmation par Contraintes(JFPC'13)*, 2013. 117, 122

[93] Charles Prud'homme, Xavier Lorca, Rémi Douence, and Narendra Jussien. Propagation engine prototyping with a domain specific language. *Constraints*, 19(1) :57–76, 2014. 117, 122

[94] Charles Prud'homme, Xavier Lorca, and Narendra Jussien. Explanation-guided large neighborhood search. In *CP Doctoral Program 2013*, pages 25–30, `http://cp2013.a4cp.org/sites/default/files/uploads/dp-2013-proceedings.pdf`, 2013. 81, 122

[95] Charles Prud'homme, Xavier Lorca, and Narendra Jussien. Explanation-based large neighborhood search. *Constraints*, 2014. 81, 122

[96] Philippe Refalo. Impact-based search strategies for constraint programming. In *CP*, pages 557–571, 2004. 14

[97] Jean-Charles Régin. Ac-* : A configurable, generic and adaptive arc consistency algorithm. In *CP*, pages 505–519, 2005. 24

[98] Andrea Roli, Stefano Benedettini, Thomas Stützle, and Christian Blum. Large neighbourhood search algorithms for the founder sequence reconstruction problem. *Computers & operations research*, 39(2) :213–224, 2012. 36

[99] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, NY, USA, 2006. 19

[100] Jean-Charles Régin. A filtering algorithm for constraints of difference in csps. In Barbara Hayes-Roth and Richard E. Korf, editors, *AAAI*, pages 362–367. AAAI Press / The MIT Press, 1994. 25, 26, 28, 29

[101] T. Schiex and G. Verfaillie. Nogood recording for static and dynamic constraint satisfaction problem. *IJAIT*, 3(2) :187–207, 1994. 37

[102] Tom Schrijvers, Guido Tack, Pieter Wuille, Horst Samulowitz, and Peter J. Stuckey. Search combinators. *CoRR*, abs/1203.1095, 2012. 14, 85

[103] Christian Schulte. Comparing trailing and copying for constraint programming. In *ICLP*, pages 275–289, 1999. 32

[104] Christian Schulte and Mats Carlsson. Finite domain constraint programming systems. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, Foundations of Artificial Intelligence, chapter 14, pages 495–526. Elsevier Science Publishers, Amsterdam, The Netherlands, 2006. 19, 20, 26, 27

[105] Christian Schulte and Peter J. Stuckey. Speeding up constraint propagation. In *CP*, pages 619–633, 2004. 29, 30

[106] Christian Schulte and Peter J. Stuckey. Efficient constraint propagation engines. *ACM Trans. Program. Lang. Syst.*, 31(1), 2008. 29, 30, 90, 92, 96, 105

[107] Christian Schulte and Guido Tack. Implementing efficient propagation control. In *Proceedings of TRICS : Techniques foR Implementing Constraint programming Systems, a conference workshop of CP 2010*, St Andrews, UK, sep 2010. 29, 30

[108] Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In Michael Maher and Jean-Francois Puget, editors, *Principles and Practice of Constraint Programming at CP98*, volume 1520 of *Lecture Notes in Computer Science*, pages 417–431. Springer Berlin Heidelberg, 1998. 34, 35, 36, 47

[109] Christian Shulte, Guido Tack, and Mikael Z. Lagerkvist. Gecode : Generic Constraint Development Environment. `http://www.gecode.org`. Accès : 15 Novembre 2013. 86, 122

[110] Efstathios Stamatatos and Kostas Stergiou. Learning how to propagate using random probing. In Willem-Jan Hoeve and JohnN. Hooker, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 5547 of *Lecture Notes in Computer Science*, pages 263–278. Springer Berlin Heidelberg, 2009. 30

[111] Peter J. Stuckey. Lazy clause generation : Combining the power of sat and cp (and mip ?) solving. In *CPAIOR*, pages 5–9, 2010. 42, 59

[112] Juha-Pekka Tolvanen, Jonathan Sprinkle, Matti Rossi, and Steven Kelly. The 10th workshop on domain-specific modeling. In *SPLASH/OOPSLA Companion*, pages 269–270, 2010. 89

[113] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages : an annotated bibliography. *SIGPLAN Not.*, 35(6) :26–36, 2000. 89

[114] Gérard Verfaillie and Narendra Jussien. Constraint solving in uncertain and dynamic environments – a survey. *Constraints*, 10(3) :253–281, 2005. 37

[115] Julien Vion. Heuristique de révision et contraintes hétérogènes. In Simon de Givry, editor, *Huitièmes Journées Francophones de Programmation par Contraintes - JFPC 2012*, Toulouse, France, May 2012. 30

[116] Toby Walsh. Depth-bounded discrepancy search. In *IJCAI*, pages 1388–1395, 1997. 33

[117] David A. Watt. *Programming language concepts and paradigms*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990. 89

[118] Yuanlin Zhang and Roland H. C. Yap. Making ac-3 an optimal algorithm. In *IJCAI*, pages 316–321, 2001. 24

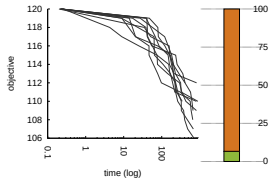
Annexes

Graphiques détaillés des voisinages pour la LNS

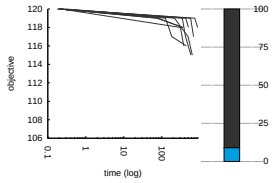
Dans cette section, nous reportons les graphiques illustrant l'évolution de l'objectif et la répartition des voisinages pour toutes les instances traitées dans le chapitre 5.

Pour rappel, chaque figure présentée dans cette section est composée d'un graphique et d'un histogramme. Le graphique illustre l'évolution de la valeur de la variable objectif tout au long de la résolution (dans une échelle logarithmique) des dix exécutions de la même approche (PGLNS, cftLNS, objLNS, EBLNS et PaEGLNS). L'histogramme reporte la répartition moyenne des voisinages du candidat utilisés pour résoudre une instance donnée. Le code couleur est le suivant :

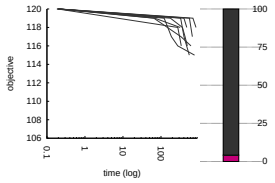
- `exp-obj` est en rose,
- `exp-cft` est en bleu,
- `ran` est en gris,
- `pgn` est en vert,
- `repgn` est en jaune et,
- `rapgn` est en orange.



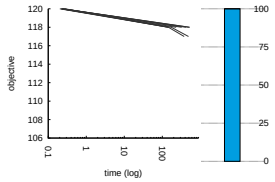
1.1 `cars_cars_4_72` with PGLNS



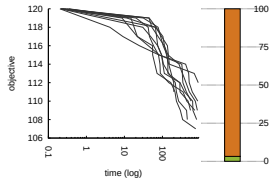
1.2 `cars_cars_4_72` with cftLNS



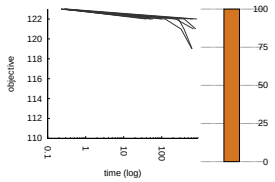
1.3 `cars_cars_4_72` with objLNS



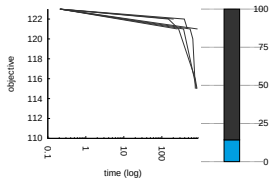
1.4 `cars_cars_4_72` with EBLNS



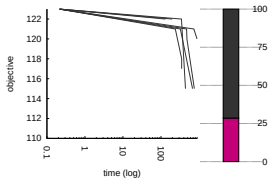
1.5 `cars_cars_4_72` with PaEGLNS



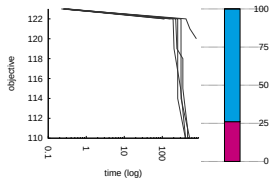
1.6 `cars_cars_16_81` with PGLNS



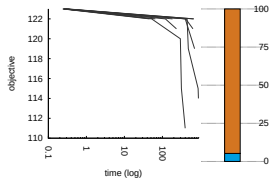
1.7 `cars_cars_16_81` with cftLNS



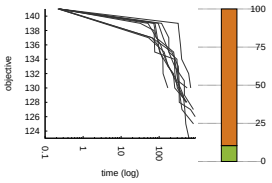
1.8 `cars_cars_16_81` with objLNS



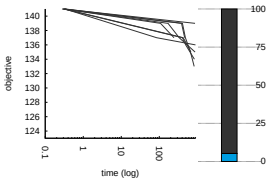
1.9 `cars_cars_16_81` with EBLNS



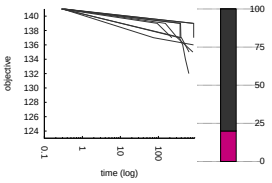
1.10 `cars_cars_16_81` with PaEGLNS



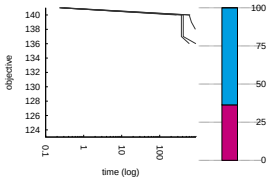
1.11 `cars_cars_26_82` with PGLNS



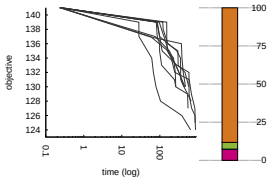
1.12 `cars_cars_26_82` with cftLNS



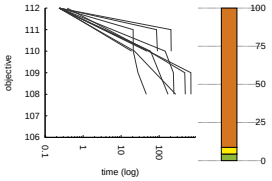
1.13 `cars_cars_26_82` with objLNS



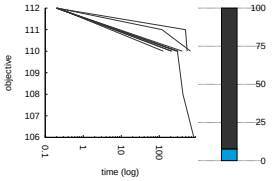
1.14 `cars_cars_26_82` with EBLNS



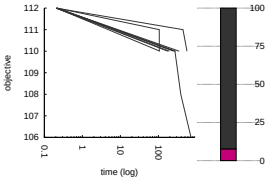
1.15 `cars_cars_26_82` with PaEGLNS



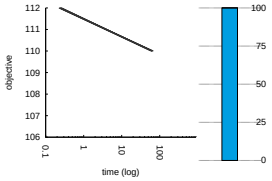
1.16 `cars_cars_41_66` with PGLNS



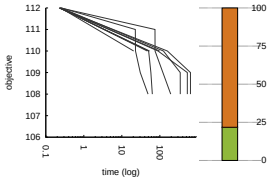
1.17 `cars_cars_41_66` with cftLNS



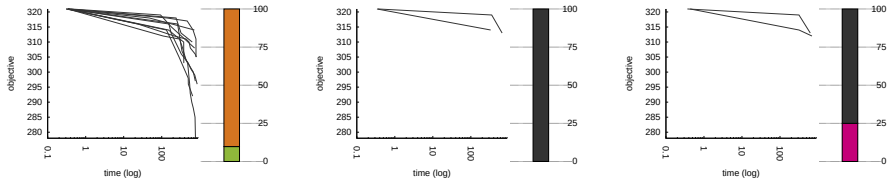
1.18 `cars_cars_41_66` with objLNS



1.19 `cars_cars_41_66` with EBLNS



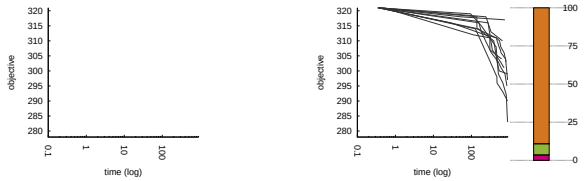
1.20 `cars_cars_41_66` with PaEGLNS



1.21 `cars_cars_90_05` with PGLNS

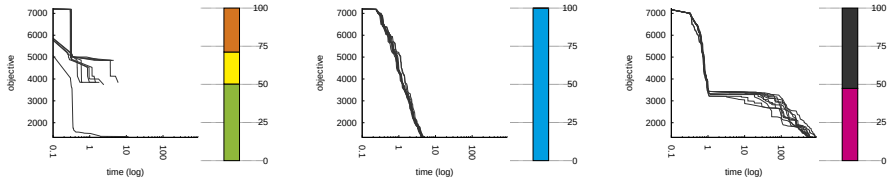
1.22 `cars_cars_90_05` with cftLNS

1.23 `cars_cars_90_05` with objLNS



1.24 `cars_cars_90_05` with EBLNS

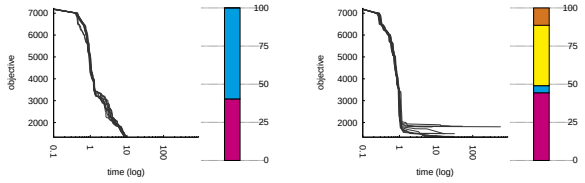
1.25 `cars_cars_90_05` with PaEGLNS



1.26 `fastfood_ff3` with PGLNS

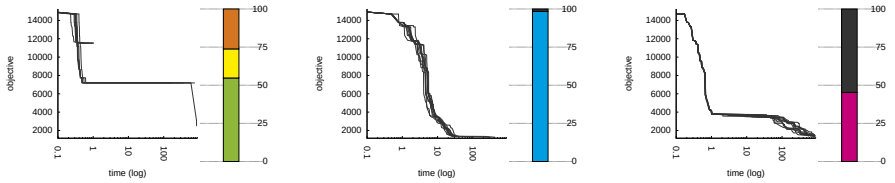
1.27 `fastfood_ff3` with cftLNS

1.28 `fastfood_ff3` with objLNS



1.29 `fastfood_ff3` with EBLNS

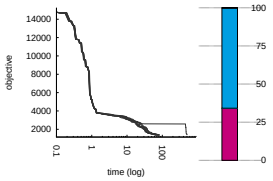
1.30 `fastfood_ff3` with PaEGLNS



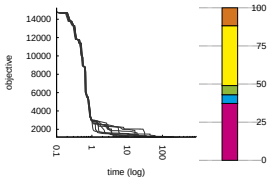
1.31 `fastfood_ff58` with PGLNS

1.32 `fastfood_ff58` with cftLNS

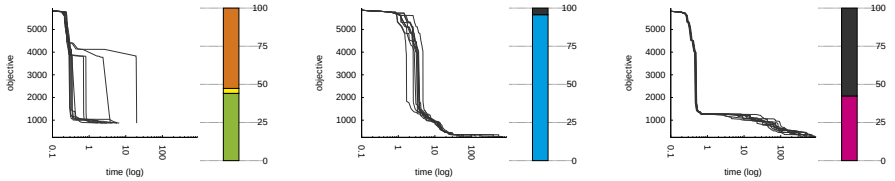
1.33 `fastfood_ff58` with objLNS



1.34 `fastfood_ff58` with EBLNS



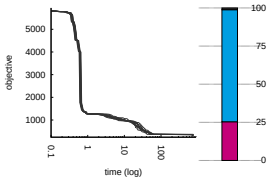
1.35 `fastfood_ff58` with PaEGLNS



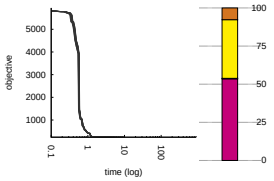
1.36 `fastfood_ff59` with PGLNS

1.37 `fastfood_ff59` with cftLNS

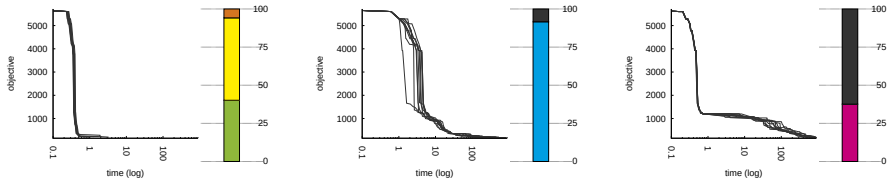
1.38 `fastfood_ff59` with objLNS



1.39 `fastfood_ff59` with EBLNS



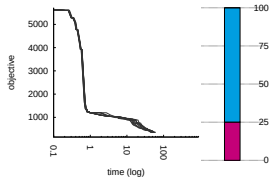
1.40 `fastfood_ff59` with PaEGLNS



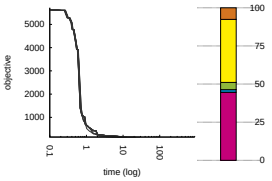
1.41 `fastfood_ff61` with PGLNS

1.42 `fastfood_ff61` with cftLNS

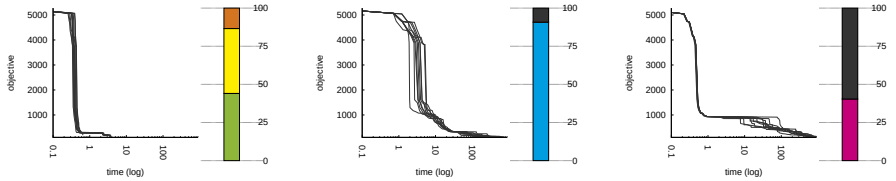
1.43 `fastfood_ff61` with objLNS



1.44 `fastfood_ff61` with EBLNS



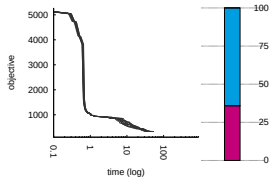
1.45 `fastfood_ff61` with PaEGLNS



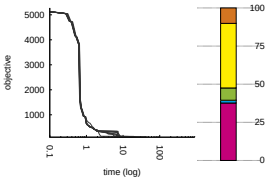
1.46 `fastfood_ff63` with PGLNS

1.47 `fastfood_ff63` with cftLNS

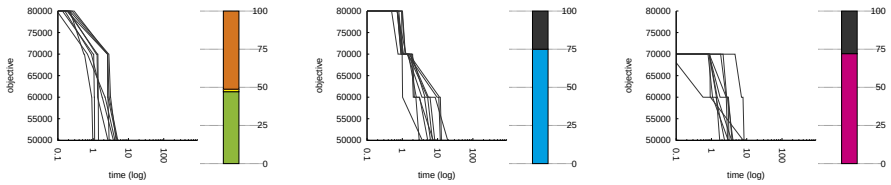
1.48 `fastfood_ff63` with objLNS



1.49 `fastfood_ff63` with EBLNS



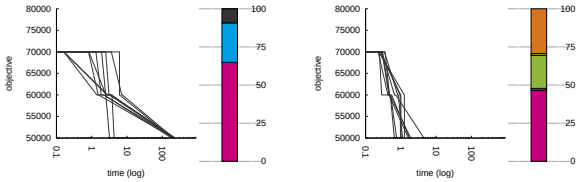
1.50 `fastfood_ff63` with PaEGLNS



1.51 `league_model20-3-5` with PGLNS

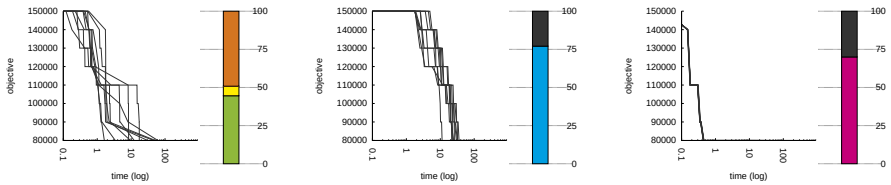
1.52 `league_model20-3-5` with cftLNS

1.53 `league_model20-3-5` with objLNS



1.54 `league_model20-3-5` with EBLNS

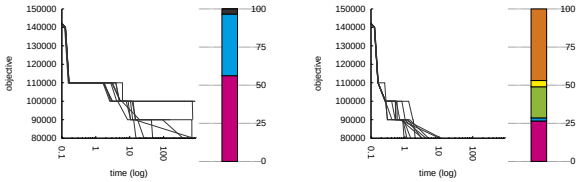
1.55 `league_model20-3-5` with PaEGLNS



1.56 `league_model30-4-6` with PGLNS

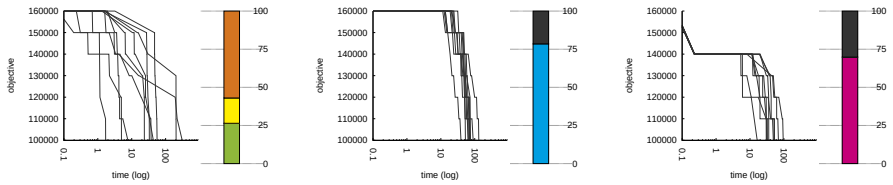
1.57 `league_model30-4-6` with cftLNS

1.58 `league_model30-4-6` with objLNS



1.59 `league_model30-4-6` with EBLNS

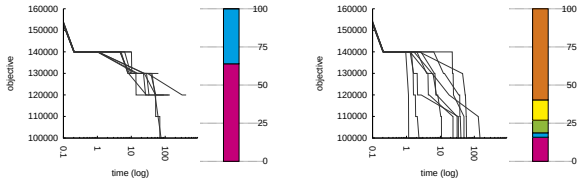
1.60 `league_model30-4-6` with PaEGLNS



1.61 `league_model50-4-4` with PGLNS

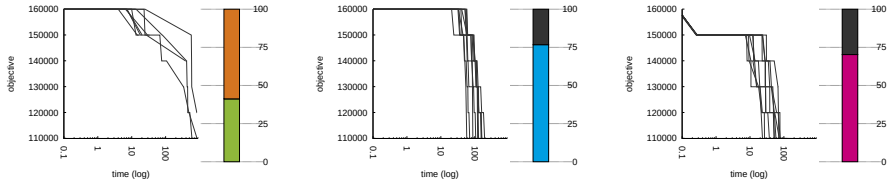
1.62 `league_model50-4-4` with cftLNS

1.63 `league_model50-4-4` with objLNS



1.64 `league_model50-4-4` with EBLNS

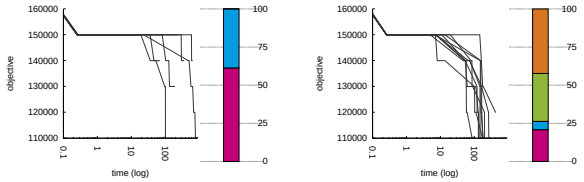
1.65 `league_model50-4-4` with PaEGLNS



1.66 `league_model55-3-12` with PGLNS

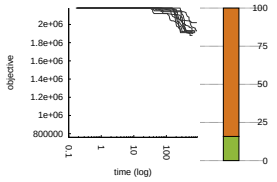
1.67 `league_model55-3-12` with cftLNS

1.68 `league_model55-3-12` with objLNS

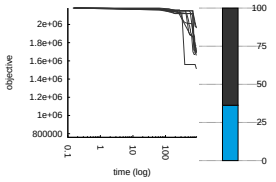


1.69 `league_model55-3-12` with EBLNS

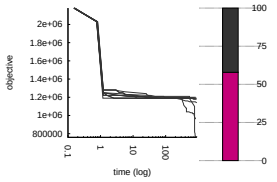
1.70 `league_model55-3-12` with PaEGLNS



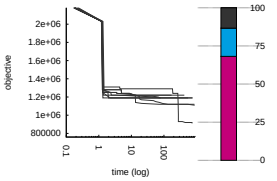
1.71 league_model90-18-20 with PGLNS



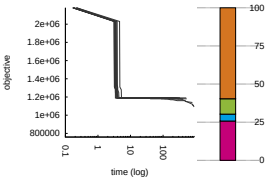
1.72 league_model90-18-20 with cftLNS



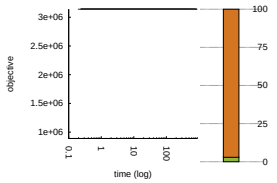
1.73 league_model90-18-20 with objLNS



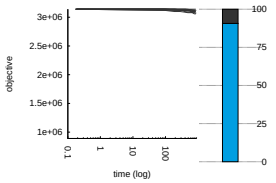
1.74 league_model90-18-20 with EBLNS



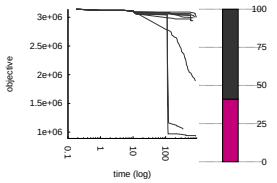
1.75 league_model90-18-20 with PaEGLNS



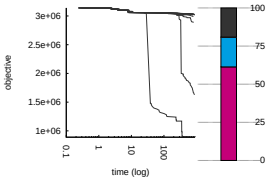
1.76 league_model100-21-12 with PGLNS



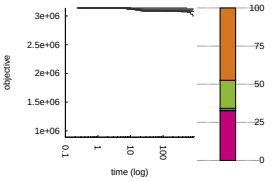
1.77 league_model100-21-12 with cftLNS



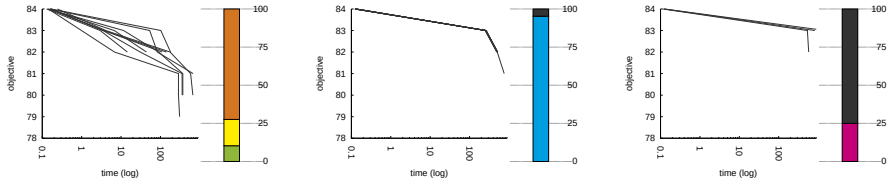
1.78 league_model100-21-12 with objLNS



1.79 league_model100-21-12 with EBLNS



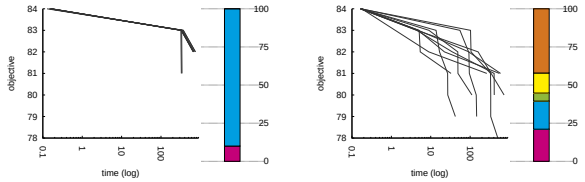
1.80 league_model100-21-12 with PaEGLNS



1.81 `rcpsp_11` with PGLNS

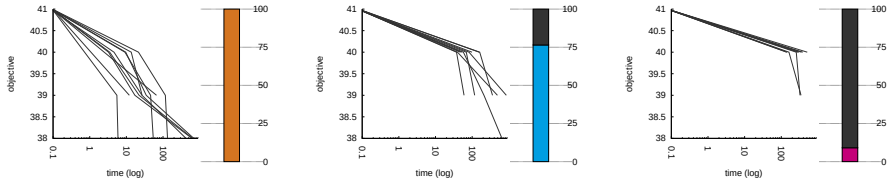
1.82 `rcpsp_11` with cftLNS

1.83 `rcpsp_11` with objLNS



1.84 `rcpsp_11` with EBLNS

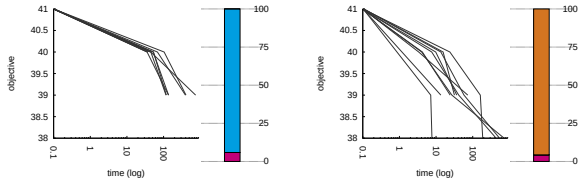
1.85 `rcpsp_11` with PaEGLNS



1.86 `rcpsp_12` with PGLNS

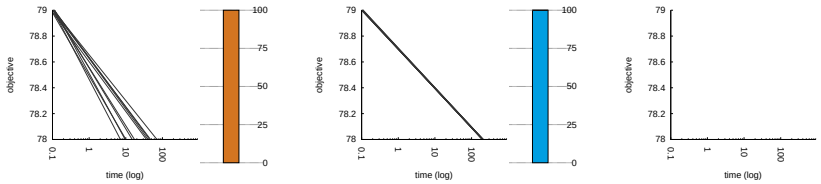
1.87 `rcpsp_12` with cftLNS

1.88 `rcpsp_12` with objLNS



1.89 `rcpsp_12` with EBLNS

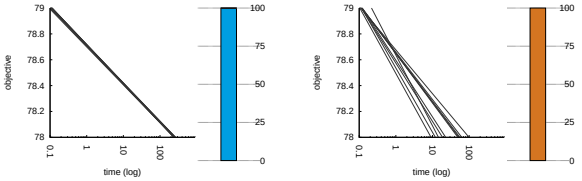
1.90 `rcpsp_12` with PaEGLNS



1.91 `rcpsp_13` with PGLNS

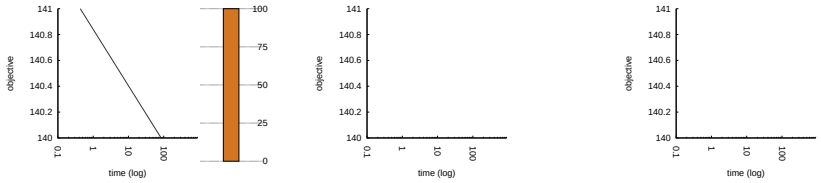
1.92 `rcpsp_13` with cftLNS

1.93 `rcpsp_13` with objLNS



1.94 `rcpsp_13` with EBLNS

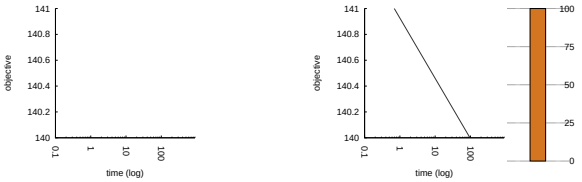
1.95 `rcpsp_13` with PaEGLNS



1.96 `rcpsp_14` with PGLNS

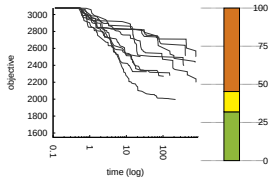
1.97 `rcpsp_14` with cftLNS

1.98 `rcpsp_14` with objLNS

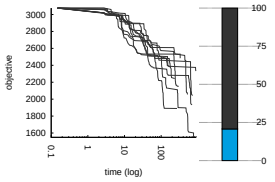


1.99 `rcpsp_14` with EBLNS

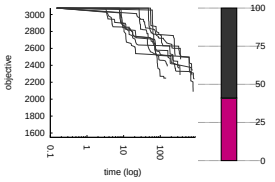
1.100 `rcpsp_14` with PaEGLNS



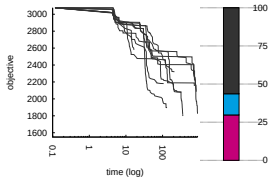
1.101
`vrp_A-n38-k5.vrp`
with PGLNS



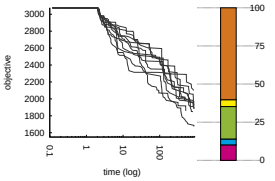
1.102
`vrp_A-n38-k5.vrp`
with cftLNS



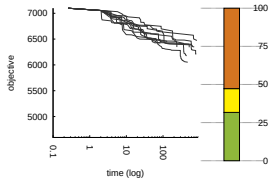
1.103
`vrp_A-n38-k5.vrp`
with objLNS



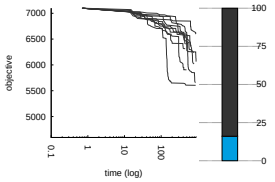
1.104
`vrp_A-n38-k5.vrp`
with EBLNS



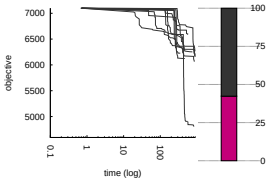
1.105
`vrp_A-n38-k5.vrp`
with PaEGLNS



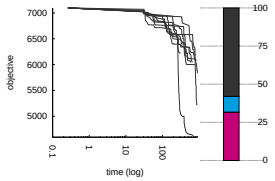
1.106
`vrp_A-n62-k8.vrp`
with PGLNS



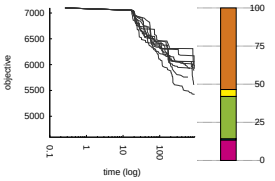
1.107
`vrp_A-n62-k8.vrp`
with cftLNS



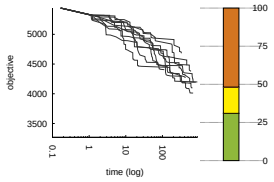
1.108
`vrp_A-n62-k8.vrp`
with objLNS



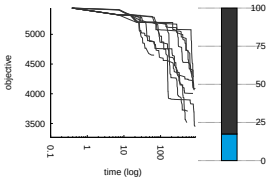
1.109
`vrp_A-n62-k8.vrp`
with EBLNS



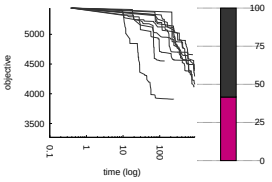
1.110
`vrp_A-n62-k8.vrp`
with PaEGLNS



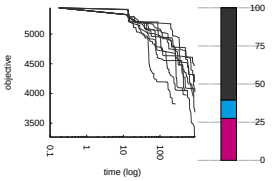
1.111
`vrp_B-n51-k7.vrp`
with PGLNS



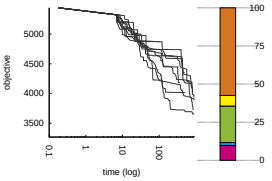
1.112
`vrp_B-n51-k7.vrp`
with cftLNS



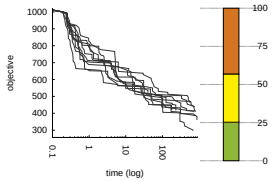
1.113
`vrp_B-n51-k7.vrp`
with objLNS



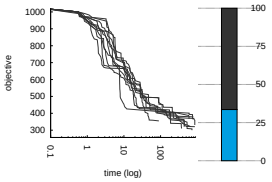
1.114
`vrp_B-n51-k7.vrp`
with EBLNS



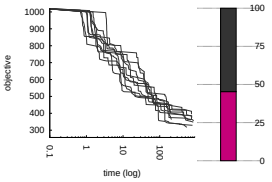
1.115
`vrp_B-n51-k7.vrp`
with PaEGLNS



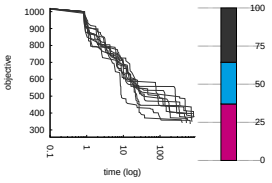
1.116
`vrp_P-n20-k2.vrp`
with PGLNS



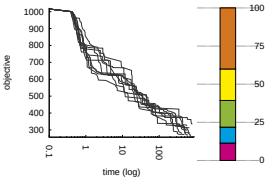
1.117
`vrp_P-n20-k2.vrp`
with cftLNS



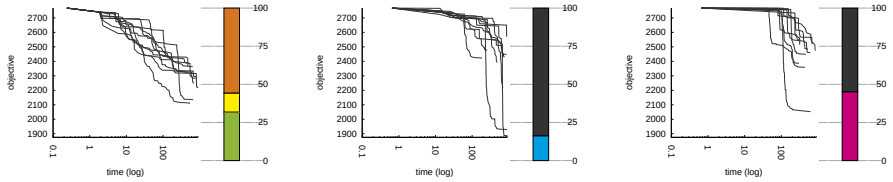
1.118
`vrp_P-n20-k2.vrp`
with objLNS



1.119
`vrp_P-n20-k2.vrp`
with EBLNS



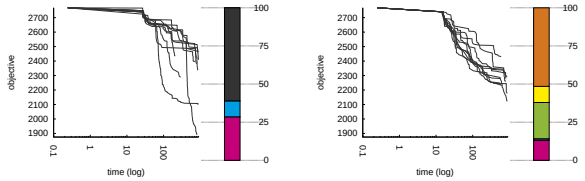
1.120
`vrp_P-n20-k2.vrp`
with PaEGLNS



1.121
`vrp_P-n60-k15.vrp`
with PGLNS

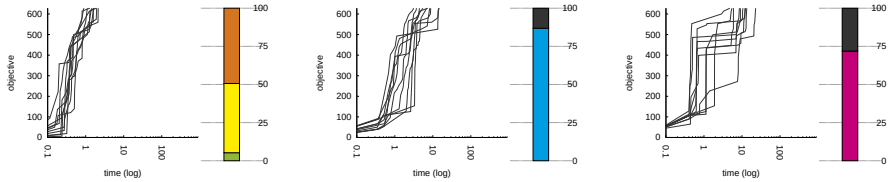
1.122
`vrp_P-n60-k15.vrp`
with cftLNS

1.123
`vrp_P-n60-k15.vrp`
with objLNS



1.124
`vrp_P-n60-k15.vrp`
with EBLNS

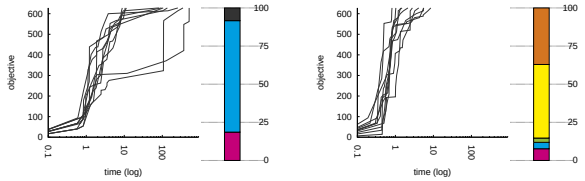
1.125
`vrp_P-n60-k15.vrp`
with PaEGLNS



1.126
`mario_mario_easy_2`
with PGLNS

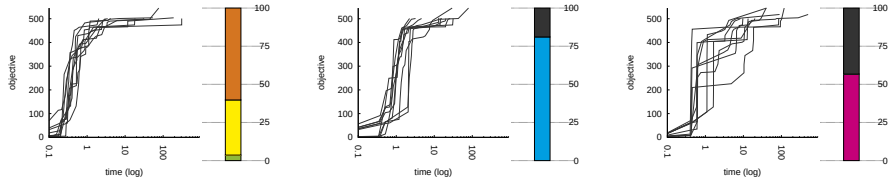
1.127
`mario_mario_easy_2`
with cftLNS

1.128
`mario_mario_easy_2`
with objLNS



1.129
`mario_mario_easy_2`
with EBLNS

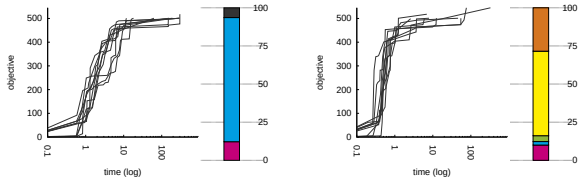
1.130
`mario_mario_easy_2`
with PaEGLNS



1.131 `mario_mario_easy_4` with PGLNS

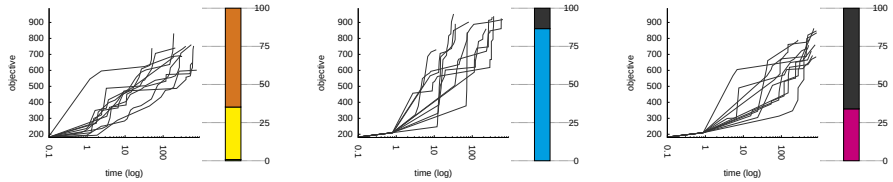
1.132 `mario_mario_easy_4` with cftLNS

1.133 `mario_mario_easy_4` with objLNS



1.134 `mario_mario_easy_4` with EBLNS

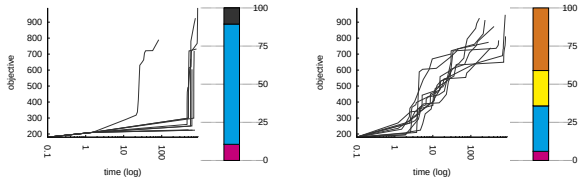
1.135 `mario_mario_easy_4` with PaEGLNS



1.136 `mario_mario_n_medium_2` with PGLNS

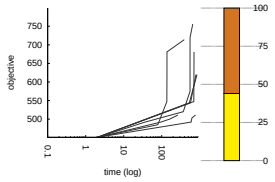
1.137 `mario_mario_n_medium_2` with cftLNS

1.138 `mario_mario_n_medium_2` with objLNS

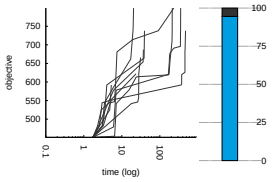


1.139 `mario_mario_n_medium_2` with EBLNS

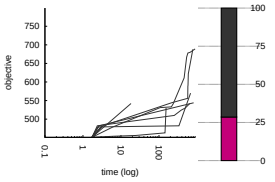
1.140 `mario_mario_n_medium_2` with PaEGLNS



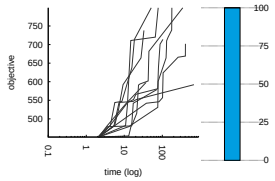
1.141 `mario_mario_n_medium_4` with PGLNS



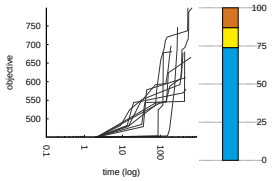
1.142 `mario_mario_n_medium_4` with cftLNS



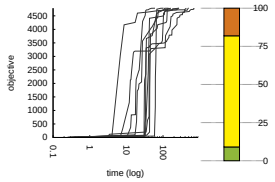
1.143 `mario_mario_n_medium_4` with objLNS



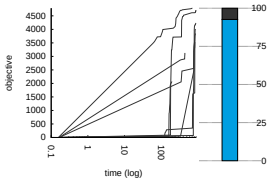
1.144 `mario_mario_n_medium_4` with EBLNS



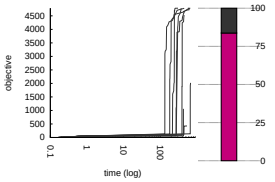
1.145 `mario_mario_n_medium_4` with PaEGLNS



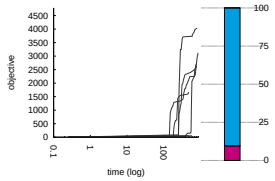
1.146 `mario_mario_t_hard_1` with PGLNS



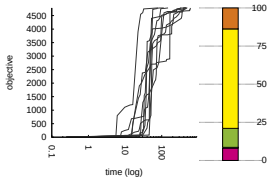
1.147 `mario_mario_t_hard_1` with cftLNS



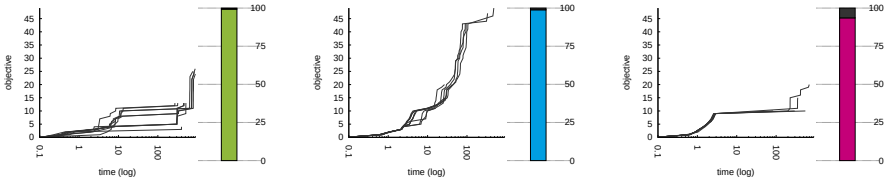
1.148 `mario_mario_t_hard_1` with objLNS



1.149 `mario_mario_t_hard_1` with EBLNS



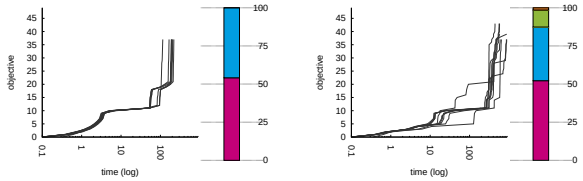
1.150 `mario_mario_t_hard_1` with PaEGLNS



1.151 pattern_set_mining_k1_ionosphere with PGLNS

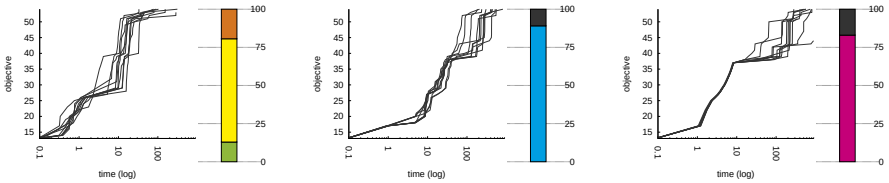
1.152 pattern_set_mining_k1_ionosphere with cftLNS

1.153 pattern_set_mining_k1_ionosphere with objLNS



1.154 pattern_set_mining_k1_ionosphere with EBLNS

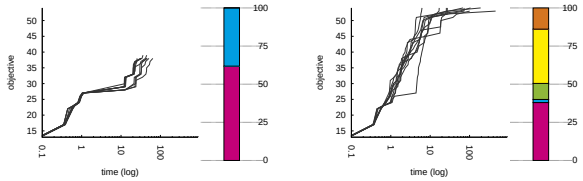
1.155 pattern_set_mining_k1_ionosphere with PaEGLNS



1.156 pattern_set_mining_k2_audiology with PGLNS

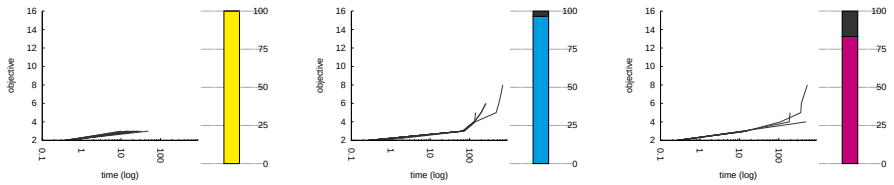
1.157 pattern_set_mining_k2_audiology with cftLNS

1.158 pattern_set_mining_k2_audiology with objLNS



1.159 pattern_set_mining_k2_audiology with EBLNS

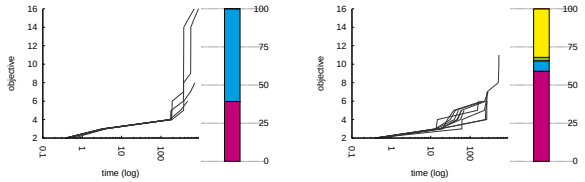
1.160 pattern_set_mining_k2_audiology with PaEGLNS



1.161 1.162 1.163

`pattern_set_mining_k2_german-credit` `pattern_set_mining_k2_german-credit` `pattern_set_mining_k2_german-credit`

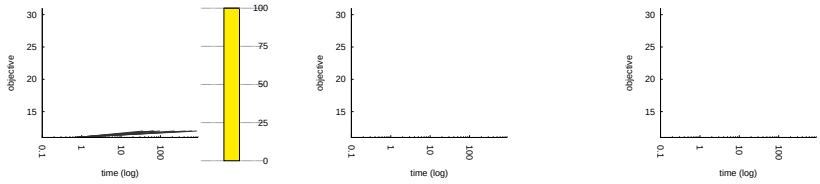
with PGLNS with cftLNS with objLNS



1.164 1.165

`pattern_set_mining_k2_german-credit` `pattern_set_mining_k2_german-credit`

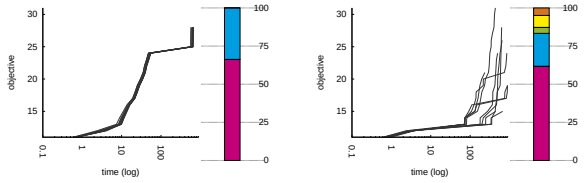
with EBLNS with PaEGLNS



1.166 1.167 1.168

`pattern_set_mining_k2_segment` `pattern_set_mining_k2_segment` `pattern_set_mining_k2_segment`

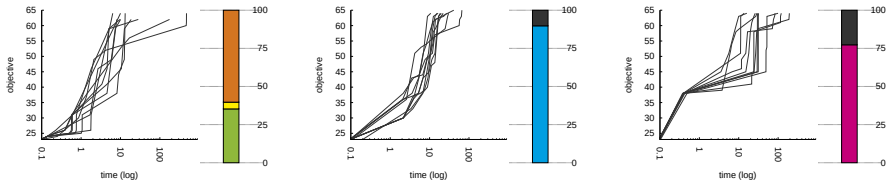
with PGLNS with cftLNS with objLNS



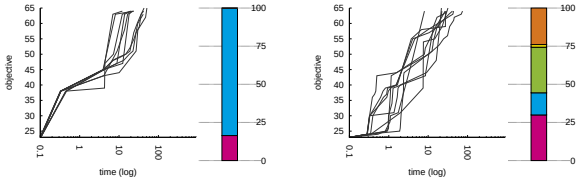
1.169 1.170

`pattern_set_mining_k2_segment` `pattern_set_mining_k2_segment`

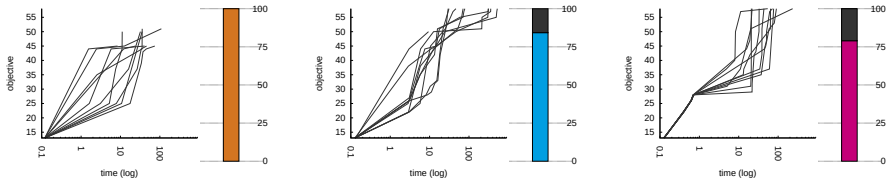
with EBLNS with PaEGLNS



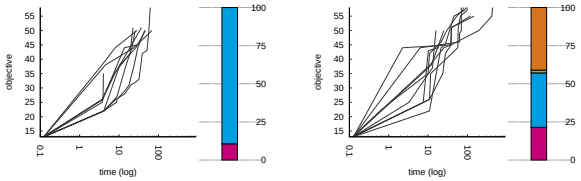
1.171 `pc_25-5-5-9` with PGLNS | 1.172 `pc_25-5-5-9` with cftLNS | 1.173 `pc_25-5-5-9` with objLNS



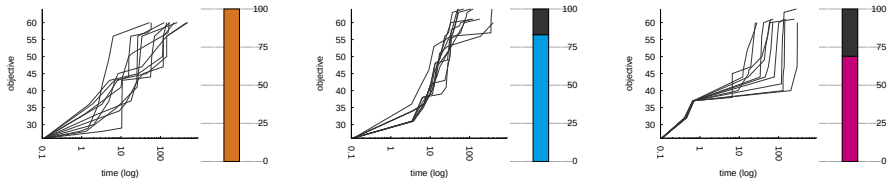
1.174 `pc_25-5-5-9` with EBLNS | 1.175 `pc_25-5-5-9` with PaEGLNS



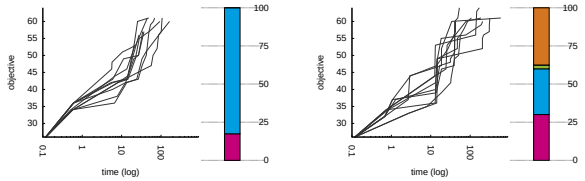
1.176 `pc_28-4-7-4` with PGLNS | 1.177 `pc_28-4-7-4` with cftLNS | 1.178 `pc_28-4-7-4` with objLNS



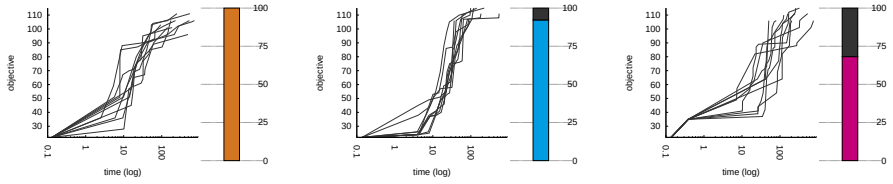
1.179 `pc_28-4-7-4` with EBLNS | 1.180 `pc_28-4-7-4` with PaEGLNS



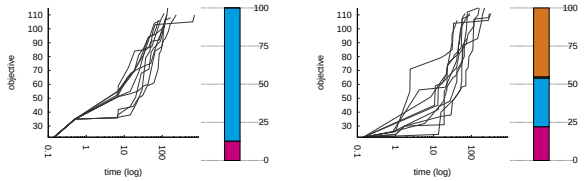
1.181 `pc_30-5-6-7` with PGLNS 1.182 `pc_30-5-6-7` with cftLNS 1.183 `pc_30-5-6-7` with objLNS



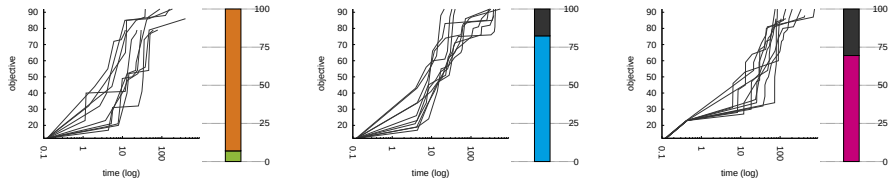
1.184 `pc_30-5-6-7` with EBLNS 1.185 `pc_30-5-6-7` with PaEGLNS



1.186 `pc_32-4-8-0` with PGLNS 1.187 `pc_32-4-8-0` with cftLNS 1.188 `pc_32-4-8-0` with objLNS



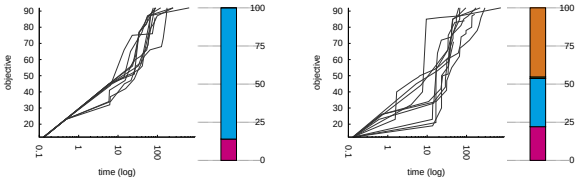
1.189 `pc_32-4-8-0` with EBLNS 1.190 `pc_32-4-8-0` with PaEGLNS



1.191 `pc_32-4-8-2` with PGLNS

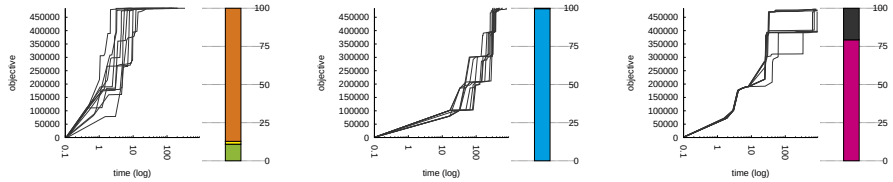
1.192 `pc_32-4-8-2` with cftLNS

1.193 `pc_32-4-8-2` with objLNS



1.194 `pc_32-4-8-2` with EBLNS

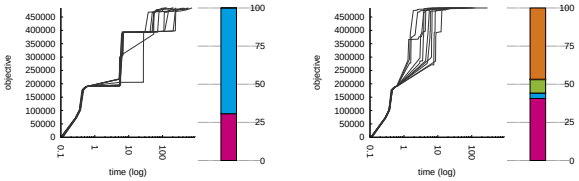
1.195 `pc_32-4-8-2` with PaEGLNS



1.196 `ship-schedule.cp_5Ships` with PGLNS

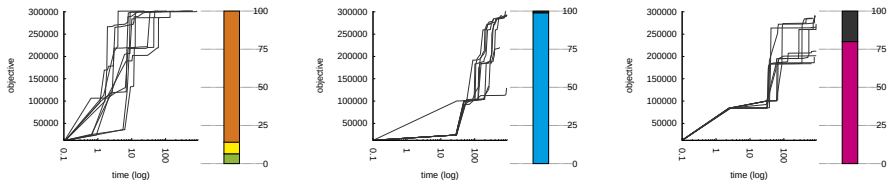
1.197 `ship-schedule.cp_5Ships` with cftLNS

1.198 `ship-schedule.cp_5Ships` with objLNS



1.199 `ship-schedule.cp_5Ships` with EBLNS

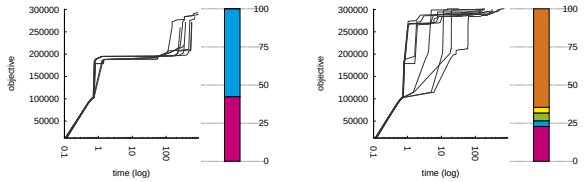
1.200 `ship-schedule.cp_5Ships` with PaEGLNS



1.201 `ship-schedule.cp_6ShipsMixed` with PGLNS

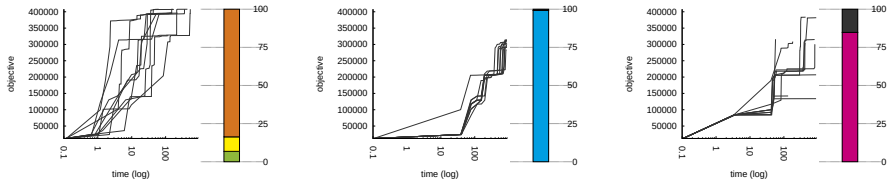
1.202 `ship-schedule.cp_6ShipsMixed` with cftLNS

1.203 `ship-schedule.cp_6ShipsMixed` with objLNS



1.204 `ship-schedule.cp_6ShipsMixed` with EBLNS

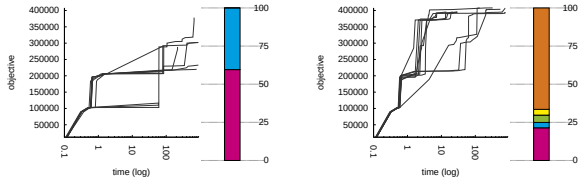
1.205 `ship-schedule.cp_6ShipsMixed` with PaEGLNS



1.206 `ship-schedule.cp_7ShipsMixed` with PGLNS

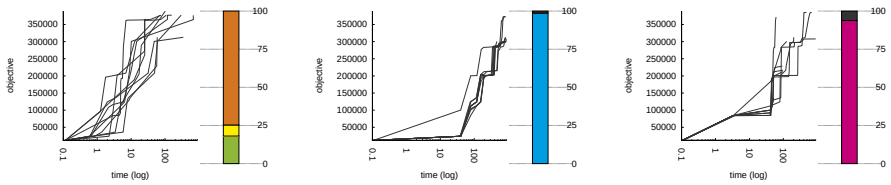
1.207 `ship-schedule.cp_7ShipsMixed` with cftLNS

1.208 `ship-schedule.cp_7ShipsMixed` with objLNS



1.209 `ship-schedule.cp_7ShipsMixed` with EBLNS

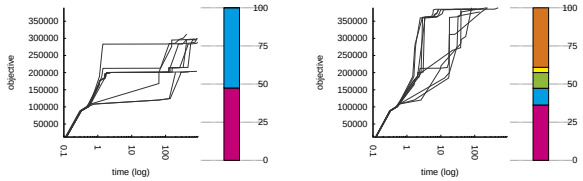
1.210 `ship-schedule.cp_7ShipsMixed` with PaEGLNS



1.211 ship-schedule.cp_7ShipsMixedUnconst with PGLNS

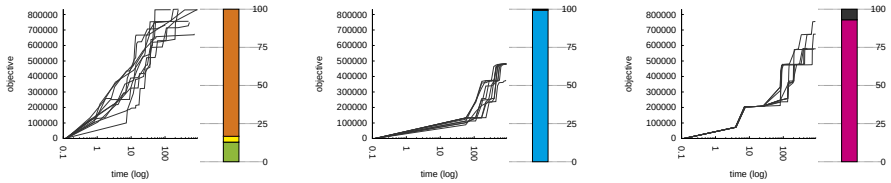
1.212 ship-schedule.cp_7ShipsMixedUnconst with cftLNS

1.213 ship-schedule.cp_7ShipsMixedUnconst with objLNS



1.214 ship-schedule.cp_7ShipsMixedUnconst with EBLNS

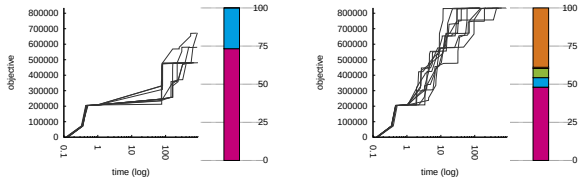
1.215 ship-schedule.cp_7ShipsMixedUnconst with PaEGLNS



1.216 ship-schedule.cp_8ShipsUnconst with PGLNS

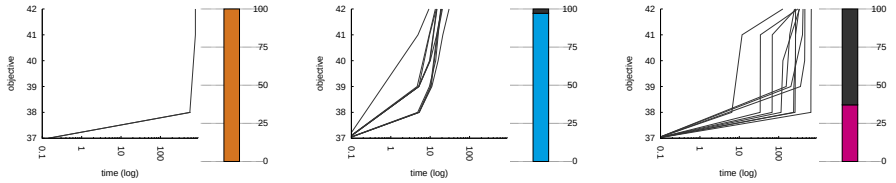
1.217 ship-schedule.cp_8ShipsUnconst with cftLNS

1.218 ship-schedule.cp_8ShipsUnconst with objLNS

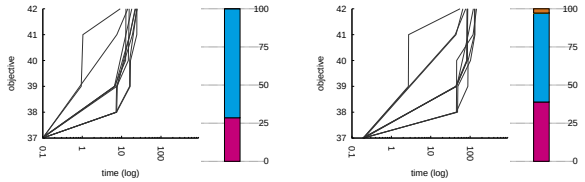


1.219 ship-schedule.cp_8ShipsUnconst with EBLNS

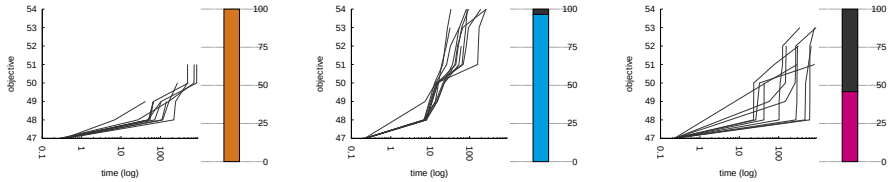
1.220 ship-schedule.cp_8ShipsUnconst with PaEGLNS



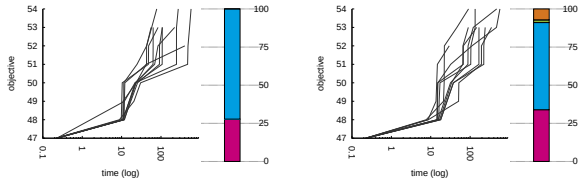
1.221 `still-life_09` with PGLNS 1.222 `still-life_09` with cftLNS 1.223 `still-life_09` with objLNS



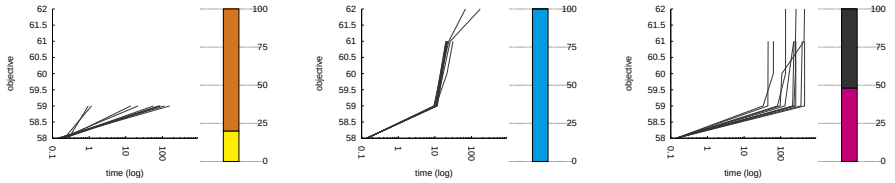
1.224 `still-life_09` with EBLNS 1.225 `still-life_09` with PaEGLNS



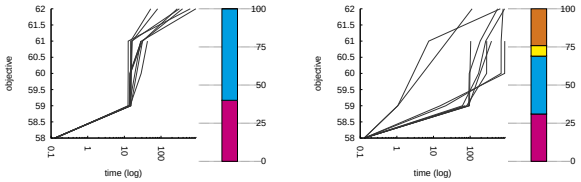
1.226 `still-life_10` with PGLNS 1.227 `still-life_10` with cftLNS 1.228 `still-life_10` with objLNS



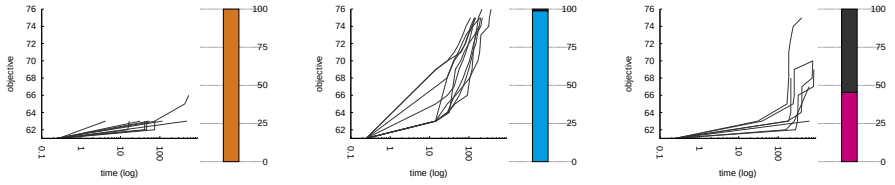
1.229 `still-life_10` with EBLNS 1.230 `still-life_10` with PaEGLNS



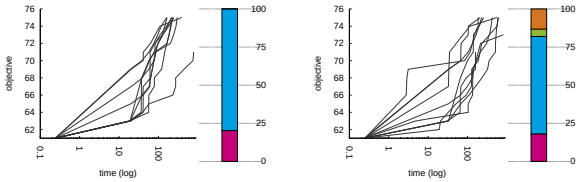
1.231 `still-life_11` with PGLNS 1.232 `still-life_11` with cftLNS 1.233 `still-life_11` with objLNS



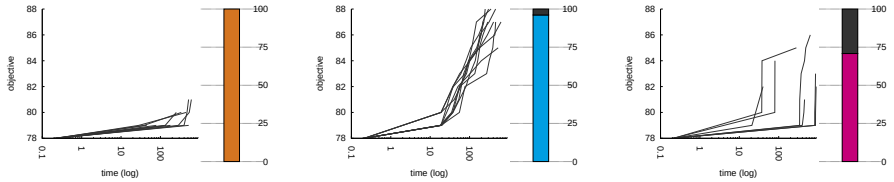
1.234 `still-life_11` with EBLNS 1.235 `still-life_11` with PaEGLNS



1.236 `still-life_12` with PGLNS 1.237 `still-life_12` with cftLNS 1.238 `still-life_12` with objLNS



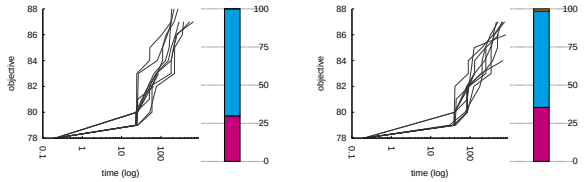
1.239 `still-life_12` with EBLNS 1.240 `still-life_12` with PaEGLNS



1.241 `still-life_13` with PGLNS

1.242 `still-life_13` with cftLNS

1.243 `still-life_13` with objLNS



1.244 `still-life_13` with EBLNS

1.245 `still-life_13` with PaEGLNS

l'université nantes angers le mans
PÔLE DE RECHERCHE ET D'ENSEIGNEMENT SUPÉRIEUR

MINES Nantes
Institut Mines-Télécom

Thèse de Doctorat

Charles PRUD'HOMME

Contrôle de la Propagation et de la Recherche dans un Solveur de Contraintes

Controlling Propagation and Search within a Constraint Solver

Résumé

La programmation par contraintes est souvent décrite, utopiquement, comme un paradigme déclaratif dans lequel l'utilisateur décrit son problème et le solveur le résout. Bien entendu, la réalité des solveurs de contraintes est plus complexe, et les besoins de personnalisation des techniques de modélisation et de résolution évoluent avec le degré d'expertise des utilisateurs. Cette thèse porte sur l'enrichissement de l'arsenal des techniques disponibles dans les solveurs de contraintes.

D'une part, nous étudions la contribution d'un système d'explications à l'exploration de l'espace de recherche, dans le cadre spécifique d'une recherche locale. Deux heuristiques de voisinages génériques exploitant singulièrement les explications sont décrites. La première se base sur la difficulté de réparer une solution partiellement détruite, la seconde repose sur la nature non-optimale de la solution courante. Ces heuristiques mettent à jour la structure interne des problèmes traités pour construire des voisins de bonne qualité pour une recherche à voisinage large. Elles sont complémentaires d'autres heuristiques de voisinages génériques, avec lesquels elles peuvent être combinées efficacement. De plus, nous proposons de rendre le système d'explications paresseux afin d'en minimiser l'empreinte.

D'autre part, nous effectuons un état des lieux des savoir-faire relatifs aux moteurs de propagation pour les solveurs de contraintes. Ces données sont exploitées opérationnellement à travers un langage dédié qui permet de personnaliser la propagation au sein d'un solveur, en fournissant des structures d'implémentation et en définissant des points de contrôle dans le solveur. Ce langage offre des concepts de haut niveau permettant à l'utilisateur d'ignorer les détails de mise en œuvre du solveur, tout en conservant un bon niveau de flexibilité et certaines garanties. Il permet l'expression de schémas de propagation spécifiques à la structure interne de chaque problème.

La mise en œuvre et les expérimentations ont été effectués dans le solveur de contraintes Choco. Cette thèse a donné lieu à une nouvelle version de l'outil globalement plus efficace et nativement expliqué.

Mots clés : Explications, Recherche à Voisinage Large, Propagation, Language Dédié.

Abstract

Constraint programming is often described, idealistically, as a declarative paradigm in which the user describes the problem and the solver solves it. Obviously, the reality of constraint solvers is more complex, and the needs in customization of modeling and solving techniques change with the level of expertise of users. This thesis focuses on enriching the arsenal of available techniques in constraint solvers.

On the one hand, we study the contribution of an explanation system to the exploration of the search space in the specific context of a local search. Two generic neighborhood heuristics which exploit explanations singularly are described. The first one is based on the difficulty of repairing a partially destroyed solution, the second one is based on the non-optimal nature of the current solution. These heuristics discover the internal structure of the problems to build good neighbors for large neighborhood search. They are complementary to other generic neighborhood heuristics, with which they can be combined effectively. In addition, we propose to make the explanation system lazy in order to minimize its footprint.

On the other hand, we undertake an inventory of know-how relative to propagation engines of constraint solvers. These data are used operationally through a domain specific language that allows users to customize the propagation schema, providing implementation structures and defining checkpoints within the solver. This language offers high-level concepts that allow the user to ignore the implementation details, while maintaining a good level of flexibility and some guarantees. It allows the expression of propagation schemas specific to the internal structure of each problem solved.

Implementation and experiments were carried out in the Choco constraint solver, developed in this thesis. This has resulted in a new version of the overall effectiveness and natively explained tool.

Key Words: Explanations, Large Neighborhood Search, Propagation, Domain Specific Language.